AFIT/GCS/ENG/92D-13

CREATING AND MANIPULATING A DOMAIN-SPECIFIC FORMAL OBJECT BASE TO SUPPORT A DOMAIN-ORIENTED APPLICATION COMPOSITION SYSTEM

THESIS

Mary Anne Randour
Captain, USAF

AFIT/GCS/ENG/92D-13

Approved for public release; distribution unlimited

AFIT/GCS/ENG/92D-13

CREATING AND MANIPULATING A DOMAIN-SPECIFIC FORMAL OBJECT BASE TO SUPPORT A DOMAIN-ORIENTED APPLICATION COMPOSITION SYSTEM

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science

Mary Anne Randour, B.S.

Captain, USAF

December, 1992

Acknowledgements

I would like to thank my advisor, Maj David Luginbuhl, for all of his guidance, support, grammar lessons, and patience. I also want to thank Maj Bailor for his guidance, not just in this specific research, but also in how to research a problem in general. Thanks also to Dr. Potoczny for being on my committee.

I owe a lot to the rest of the research group for all of their help and support. Cindy Anderson helped me more than she'll ever know just by listening and offering solutions. Mary Boom, proving that school does not have to be all work, used her sense of humor to bring "color" to many of us. Thanks also the the rest of the group: Brad Mallare (especially for all of his "geek" humor), Rusty Baldwin, and Tim Weide. A special thanks goes to Chuck Wright who was always there to offer support (or an offer to go get ice cream and/or chocolate which was even better!). These people and everyone else who spent the summer working in the computer lab made this thesis work much more enjoyable. Finally, I would like to thank my family for all of their support, for teaching me the importance of education, and for never letting me lose sight of what is truly important in life.

Mary Anne Randour

Table of Contents

List of Figures

Abstract

This research investigated technology which enables sophisticated users to specify, generate, and maintain application software in domain-oriented terms. To realize this new technology, a development environment, called Architect, was designed and implemented. Using canonical formal specifications of domain objects, Architect rapidly composes these specifications into a software application and executes a prototype of that application as a means to demonstrate its correctness before any programming language specific code is generated. This thesis investigated populating and manipulating the formal object base required by Architect. This object base is built using a domain-specific language (DSL) which serves as an interface between the user and a domain model. The domain model describes primitive domain object classes and composition rules and is formalized via a domain modeling language. The packaging of the objects into components is defined by an architecture model which was part of a separate thesis. The Software Refinery environment was used to develop a methodology for defining DSLs for Architect and for manipulating the resulting populated object base. Architect was validated using both artificial and realistic domains and was found to be a solid foundation for an application generation system.

CREATING AND MANIPULATING A DOMAIN-SPECIFIC FORMAL OBJECT BASE TO SUPPORT A DOMAIN-ORIENTED APPLICATION COMPOSITION SYSTEM

I. Plan of Attack

1.1 Background

Currently, software is not keeping up with the rapidly increasing demand, and breakthroughs in hardware technology are out-pacing corresponding developments in software engineering. To overcome this disparity, we need to develop a methodology that allows us to generate timely, validated software that meets the user's requirements. This paradigm must incorporate existing technologies – formal methods, reusability, object-oriented methodology, and artificial intelligence (AI) – and combine them to produce more capability than any of the current techniques can produce alone.

A domain-oriented application composition system, shown in Figure 1.1, may serve as a basis for this new software engineer paradigm. This approach allows sophisticated end users, called application specialists, to compose applications within a given domain. The application specialist supplies application-specific information for a specification using terminology familiar to him, not the terminology typically used by software engineers. Eventually, this interface will be visual rather than textual, making the specification process even easier for the user. This information, combined with domain knowledge stored in the domain model, builds a specification. Part of the structure of the application specialist's input is based on the architecture model used. The proposed specification can then be executed to demonstrate its behavior, and if this prototype does not match the intended operation, the proposed specification can be changed and the prototype regenerated until it exhibits the desired functionality. This prototyping cycle is done without any additional

coding (other than the initial input) and without generating a fully coded application. When the application specialist is satisfied with the execution, he can generate a complete formal specification. This formal specification is stored in an object (or knowledge) base where it can be transformed into a target application language, such as Ada, C, or even machine level code. The benefits of this technique are obvious: end users can specify applications using a language they understand and they can develop prototypes to ensure the final specification is complete and accurate. The tool that provides the means for the application specialist to supply the application-specific information is a domain-specific language (DSL).

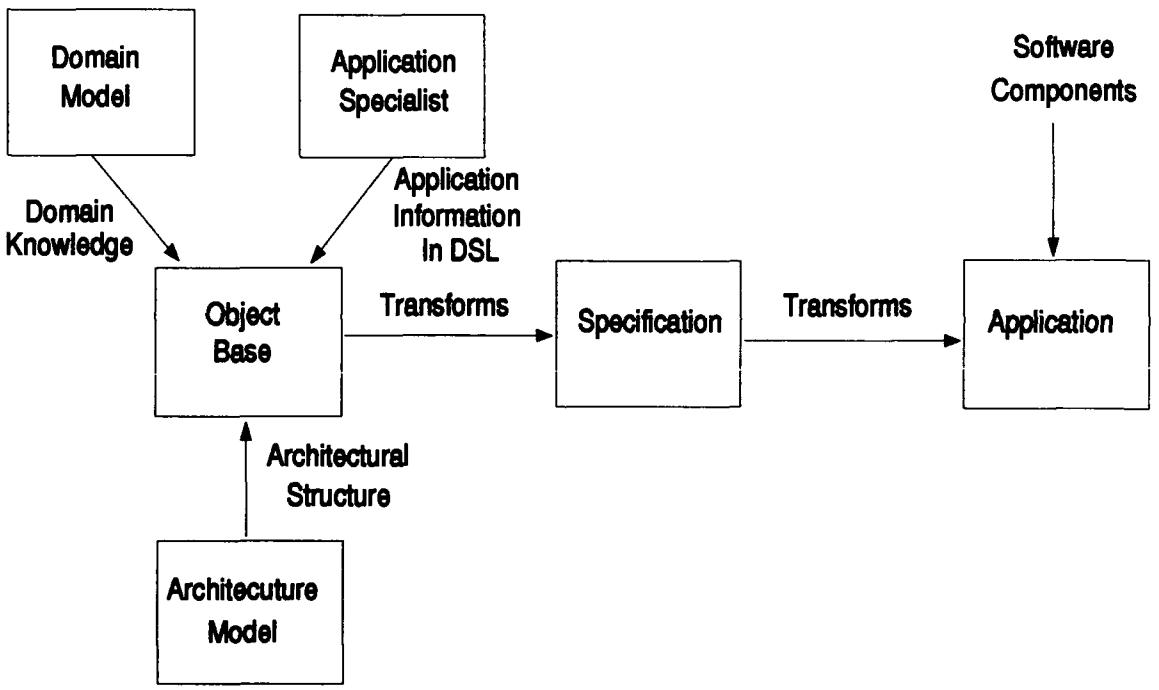

Figure 1.1. Domain-Oriented Application Composition System

Current technology cannot yet fully support this paradigm; much research is required before it can be implemented. For example, the following steps are required to generate a system capable of implementing this paradigm:

1. Define and produce domain-specific languages,
2. Develop an architectural model to define domain-specific architectures to provide an overall structure for the applications,
3. Write transformations to manipulate the object base,

4. Code software components conforming to the domain architecture and rules regarding how to combine them.

1.2 Problem

The objective of this research was to investigate the development of a system to support the paradigm described above, specifically, to create and manipulate a domain-specific formalized object base in the Software Refinery^{TM} Environment using a domain-specific language and additional REFINE functions. Other research supports other aspects of this approach; information about the implementation of software architectures can be found in (1) and details on how to visualize this process can be found in (29).

As mentioned above, a domain-specific language is one key to this new paradigm since it furnishes the interface between the user and the object base. However, developing a domain-specific language is not a trivial task. As shown in Figure 1.2, a domain-specific language must incorporate knowledge from several different disciplines. First, it must capture the domain knowledge obtained through domain analysis, including the overall software architecture (structure) of the domain. This knowledge is represented using an object-oriented methodology, modeling real-world entities as objects and operations on those objects. Following this methodology will allow users to conceptualize better the application by allowing them to represent the application as familiar, concrete objects. Also, the software components can be encapsulated so that they can be combined more easily than components partitioned functionally. Finally, general knowledge about specifications is needed so that the language can define applications. All of this knowledge must be built into a grammar, defining a domain-specific language that can be used by a language processing tool.

Proposed specifications, or application definitions, are written using the domain-specific language and parsed into the REFINE object base. The grammar for the DSL is written using the REFINE tool, DIALECT. DIALECT provides a parser that populates the abstract syntax trees (the structural representation of the object base) based on the input and the DIALECT domain model. DIALECT also provides a prettyprinter that prints the abstract syntax trees in a textual format.

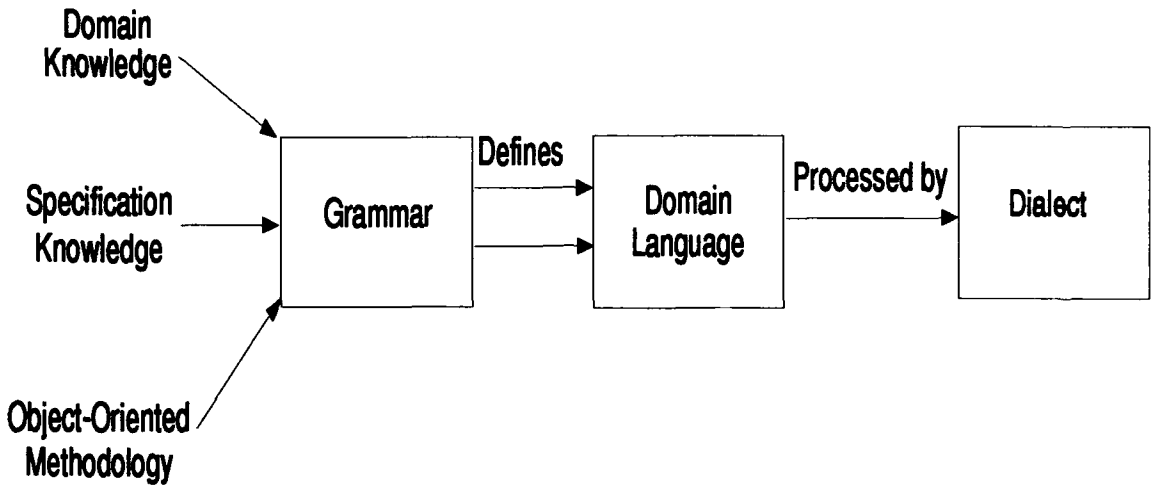

Figure 1.2. Components of a Domain-Specific Language

Getting the specification into a formalized object base is only the start (see Figure 1.3). The specification may not be completely correct as initially stated. It may violate architectural restrictions or domain restrictions, or it may not exhibit the desired behavior. Therefore the object base must be manipulated until the application definition complies with the architecture and domain constraints and displays the correct behavior.

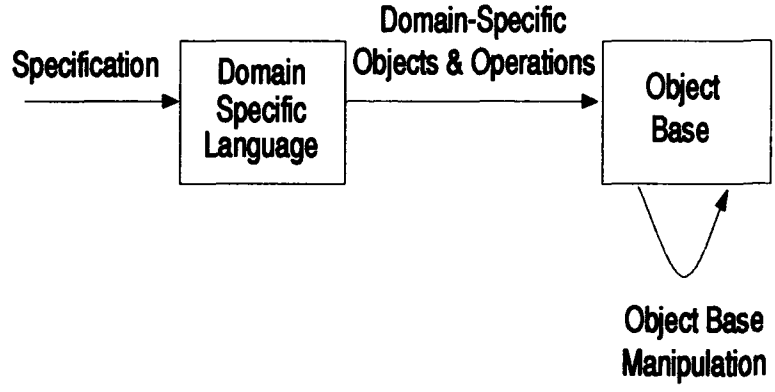

Figure 1.3. Domain-Specific Language Usage

1.3 Summary of Current Knowledge

The concept of a domain-specific language has been discussed in several technical journals and dissertations; however, very little has been written concerning the details

of exactly what a domain-specific language is. Chapter II gives an overview of related literature.

1.4 Approach

Developing a domain-specific language to populate a formalized object base requires several steps. Creating a grammar that describes the language is only one step in the process. The following were needed to develop a domain-specific language.

1. The first step was to define the term domain-specific language. Several articles in the literature discuss domain-specific languages, but only at a high level. The definition of a domain-specific language must include details not given in the current literature and the language must be defined in the context of how it fits into the system being developed as part of this research. The domain-specific language provides a front end for the user to specify the application in an object-oriented methodology so it must represent objects and their attributes and operations on those objects that are familiar to the user. A domain-specific language alone cannot define the structure of an application; it will rely on a defined architecture model to provide this framework.
2. Although domain-specific languages may not be very well defined, several requirements specification languages are. One or several of these could be useful as a starting point, we investigated them to determine if they could be used. If one of these specification languages had provided a good starting point, it would have become a starting point for our grammar, but we found none that met our needs.
3. The next step was to develop a domain-specific language grammar for an artificial domain and then a more realistic domain, that of digital circuits, to validate that the implementation met the stated requirements. We began development using an artificial domain and then moved to our realistic domain to ensure that the system remained domain-independent and to determine what further research was required.
4. Implementation of several small specifications provided a means to evaluate the domain-specific language. The resulting object base was the basis for the next test: how well the specification actually described the problem. The execution of the spec-

ification was tested as part of a separate research topic, but any problems uncovered with the domain-specific language were fixed until the domain-specific language performed correctly. This was not a pass/fail type of test like normal software testing, rather it was used to evaluate the suitability of the domain-specific language.

1.5 Order of Presentation

Information about domain-specific languages and other related topics found in the literature are explained in ChapterII. Chapter III details the requirements for the development of an application composition system of which the development of domain-specific languages is a part. This chapter also includes requirements for other research and is also contained in (1). The overall design of an application composer as well as a detailed design for this aspect of the system can be found in Chapter IV. Chapter V gives the results of implementing the validating domain. The results and conclusions of this research are in Chapter VI. Hints and guidelines for the users of this system, the software engineer and application specialist, are listed in Appendices A and B, respectively. To aid the reader in understanding the how this system works, a sample session is provided in Appendix D. Finally, the REFINE code to define the validating domain model and to implement this portion of the research are listed in Appendices C and E.

II. Literature Review

2.1 Introduction

Very little has been written on the subject of domain-specific languages (DSLs). In fact, the current literature only refers to the concepts and the potential uses of DSLs; it does not give any specific information on how to implement one. However, this does not mean that other researchers do not provide some insight into developing a DSL. This chapter reviews information found concerning DSLs.

2.2 Current Research

Several different fields relate either directly or indirectly to the use of domain languages. Some areas provide a firm concept of the term, while other areas apply similar ideas without specifically referring to "domain-specific languages." Most of the research partitions the space of all possible applications into specific areas (called domains) and follows an object-oriented methodology. The reusability field gives insight into what exactly a domain-specific language is. Since a DSL can be viewed as a specialized requirements specification language, our DSL could be based on research done in this field. A DSL cannot be isolated from the process of domain analysis, so we also must investigate this field. Because we also plan to incorporate a software architecture model, we must understand these models. Program generation and synthesis systems give examples of how a domain language could fit into an overall system.

2.2.1 Reusability One of the early pioneers in the field of applying domain analysis for the purpose of reusability was James Neighbors. In his research on the Draco approach for constructing software from reusable components, Neighbors proposed using a domain language to describe objects and operations. Neighbors also defined a domain description that includes the following components (19:41-43):

- Parser – Defines external syntax (BNF)
- Prettyprinter – Defines how to produce external syntax of domain for all possible program fragments

- Transformations – Source-to-source transformations on objects and operations in the domain
- Components – Specify semantics of domain, one for each object and operation in domain, make implementation decisions
- Procedures – Domain specific, used when transform is algorithmic in nature

In following the Draco approach, a software engineer performs domain analysis to produce domain languages which are used to specify an application. Multiple refinements, which make implementation decisions by restating the problem in other terms, can be applied. The resulting specification can then be transformed and implemented by software components (18:565-566).

Another advocate of reusability, Rubén Prieto-Díaz, defines domain analysis as the process used to identify, capture, and organize information. A domain language is one of the outputs of this process (21:47-48). He further defines a domain specific language as "a language with syntax and semantics designed to represent all valid actions and objects in a particular domain" (22:23) and "a collection of rules that relate objects and functions and which can be made explicit and encapsulated in a formal language and further used as a specification language for the construction of systems in that domain" (22:26). The domain language describes the objects and operations and becomes the framework for new specifications (22:23-24). Prieto-Díaz only describes the ideas, he does not explain how a DSL can be described.

2.2.2 Requirements Languages Often, requirements are specified in a natural language, such as English. However, several different requirements languages now exist. Sol Greenspan defined what he calls a Requirements Modeling Language (RML). RML is an object-oriented language where objects represent real-world entities. RML uses three levels of classification: tokens (instances of a class), classes (instances of one or more metaclasses), and metaclasses (top-level classes). RML also uses several abstraction mechanisms:

- Properties – Relationships between objects
- Aggregation – Collection of objects
- Class – Classification of objects that share common properties
- Instances – Specific members of a class

- Definitional Properties – General characteristics that apply to all instances of a class or metaclass

Greenspan defines several principles that must be included in all requirements languages (9:3-4):

- The languages should allow direct and natural modeling of the world. This is best accomplished using an object-oriented methodology where objects represent concepts and entities and the behavior is represented by operations on the objects.
- Requirements languages should support the organization and management of large descriptions. Greenspan recommends using abstraction principles such as aggregation, classification, and generalization to support this.
- They should allow expression of assertions (what is true in the world), entities (objects in the world), and activities (events that cause change). RML uses three kinds of objects and First-Order Logic (for the assertions) to accomplish this.
- The language must be uniform in its use of basic principles so that it will be easy to learn and use.
- Requirements languages should be formal; their features should be defined precisely.
- Finally, they must provide guidance in modeling large-scale problems.

2.2.3 Domain Analysis

2.2.3.1 Meta-Model/Meta-Language Neil Iscoe's research has focused on applying domain knowledge to a meta-model or meta-language to instantiate a domain model (10:301-302). The meta-model or meta-language is a representation structure or language used to specify the knowledge about a given domain. The formality and ability to execute the model allow reasoning and inference to support the goals of the model being developed (10:301). Iscoe defines the attributes for objects in the domain model to be: a unique name, measurement scale (including unit and granularity when appropriate), set of values, set of population parameters, initialization procedure, probability of change, and a state transition relation (11:33). Classes consist of: a set of attributes and their subsets, a set of functions used to create objects, a set of operations on the objects, a statistical summary, and a set of axioms or integrity constraints (11:77).

Iscoe also defines a hierarchical decomposition of these classes that allows for inheritance and aggregation (as an implicit part of domain structure) (11:121). Some classes are

built from previously existing ones using composition. This process uses domain knowledge to take the disjoint union of the sets of attributes, resolve any conflicts, eliminate unnecessary attributes, and add new attributes as needed (11:129).

2.2.3.2 Feature Oriented Domain Analysis In their Feature-Oriented Domain Analysis (FODA) study, the Software Engineering Institute (SEI) defines a process for domain analysis based on describing prominent or distinctive aspects of a software system, or features. This method is divided into three steps (12:4):

1. Context Analysis – Define the bounds of the domain
2. Domain Modeling – Describe the problems within the domain
3. Architecture Modeling – Create software architecture(s) to implement a solution in the problem domain.

The FODA study also defines several of the roles involved in this process. The end users and domain experts are the sources of domain information. Domain analysts organize and compile the information which is then used by requirements analysts, software designers, and end users (12:4). Another study by the SEI on Modeling Software Systems by Domains refines these definitions slightly. The roles are further described in Chapter III.

2.2.4 Software Architecture We are taking a slightly different slant to developing a DSL; we are incorporating a software architecture model. This is being done as a separate thesis (1), but is related to the development of our DSL. The model we incorporated is the Software Engineering Institute's Object Connection Update (OCU) model. The OCU model describes objects which are grouped into subsystems to define a specific software architecture. More details on the OCU model can be found in Chapter III and details on this specific implementation of the OCU model can be found in (1).

2.2.5 Program Generation David Barstow defines automatic programming as a system that allows a naive user (one who does not necessarily understand programming) to describe a problem (3). Research has investigated automatic programming systems that ultimately assist in transforming specifications into high-level language code. Currently, each system only automates different portions of the whole and each takes a different

approach, but some do share several common features that can be applied to research at AFIT. They are concerned with eliciting a formal specification from a naive user and manipulating this specification to generate an application.

2.2.5.1 Automatic Programming Technologies for Avionics Software Lockheed's Software Technology Center has developed the Automatic Programming Technologies for Avionics Software (APTAS) system. APTAS allows the user to specify an application within the target tracking domain, generates a software architecture, produces executable intermediary code, and finally generates Ada code. The user's input is based on pop-up question forms that guide the user through the specification process. APTAS is explained in greater detail in Chapter III and (16).

2.2.5.2 Hierarchical Software Systems with Reusable Components Don Batory and Sean O'Malley have developed a model of hierarchical software design based on interchangeable software components (4:2). They define components as their fundamental units of software. Components that share common interfaces, i.e., are interchangeable, belong to what they call a realm. Components are combined into systems according to specific rules of composition. In this method, a domain model is a set of realms and rules of composition. Batory and O'Malley equate this scheme to a grammar where the productions define the rules of composition and the terminals represent the components (4:4-5). This model is discussed further in Chapter III.

2.2.5.3 Other Systems Other research projects do not specifically use domain-specific languages, but they are implementations of systems similar to the paradigm described above. Howard Reubenstein and Richard Waters describe a Requirements Apprentice which assists an analyst in specifying a program. This system does not use a domain-specific language, but it does use what they call a cliché library to store information about requirements in general and the domain in particular (25). The Kestrel Interactive Development System (KIDS) provides a set of tools to transform specifications into code, but unlike the others, it includes tactics designed to preserve correctness of the program at each step of the transformation process. A domain theory, developed early in

the process, must be preserved throughout the process to ensure the final product meets the specification and does not contradict the domain knowledge (26:1025). The Knowledge-Based Requirements Acquisition System (KBRAS) is an AI toolkit that automates the gathering of software requirements by eliciting information from the user. This system is based on the idea that requirements analysis is the process of reducing domain-specific knowledge into precise statements about the application (30:40-41).

2.3 Summary

Current research does not provide any concrete answers on how to develop a domain language. However, valuable research has been done in related areas. Investigation into the fields of reusability, requirements languages, and domain analysis helped us understand what features a DSL must possess. Program generation and synthesis systems guided us in developing an overall system that uses a DSL. Ideas from these areas have been useful in the development of a domain language.

III. Requirements Analysis[1]

3.1 Introduction

The wide availability of powerful, relatively low-cost computer hardware has led to an explosion in the demand for computer software products to automate a multitude of new tasks. Using traditional methods, computer scientists and programming professionals have been unable to meet, in a timely manner, this demand for the sophisticated, large-scale, reliable software systems required for these new applications. Clearly, a new approach to software design and construction is needed.

> Software engineering will evolve into a radically changed discipline. Software will become adaptive and self-configuring, enabling end users to specify, modify and maintain their own software within restricted contexts. Software engineers will deliver knowledge-based application generators rather than unmodifiable application programs. These generators will enable an end user to interactively specify requirements in domain-oriented terms.... and then automatically generate efficient code that implements these requirements. In essence, software engineers will deliver the knowledge for generating software rather than the software itself.
>
> Although end users will communicate with these software generators in domain-oriented terms, the foundation for the technology will be formal representations... Formal languages will become the lingua franca, enabling knowledge-based components to be composed into larger systems. Formal specifications will be the interface between interactive problem acquisition components and automatic program synthesis components.
>
> Software development will evolve from an art to a true engineering discipline. Software systems will no longer be developed by handcrafting large bodies of code. Rather, as in other engineering disciplines, components will be combined and specialized through a chain of value-added enhancements. The final specializations will be done by the end user. KBSE (Knowledge Based Software Engineering) will not replace the human software engineer; rather, it will provide the means for leveraging human expertise and knowledge through automated reuse. New subdisciplines, such as domain analysis and design analysis, will emerge to formalize knowledge for use in KBSE components. (17:629-630)

[1]This chapter was co-written with Capt Cynthia Anderson and also appears in AFIT Technical Report AFIT/EN/TR-92-5 and (1).

Perhaps this vision can become a reality for selected domains, not just within the next century as Michael Lowry predicts, but within the next few years. Research is currently underway at the Air Force Institute of Technology (AFIT) to achieve such a reality. Developing a full-scale application generation system, which is capable of automatically producing efficient code to satisfy user-specified requirements presented in domain-oriented terms, is a considerable task which will require several man-years of effort. However, one element of application generation, the combining or composing of required components into the proper framework or architecture, is attainable in the near term. This chapter explores the issues involved in developing such an end-user application composer and describes one possible methodology for accomplishing it.

3.2 Operational Concept

Several roles are discussed in describing this new approach to software development, an approach where the end-user generates a software application to satisfy his requirements using the software professional's knowledge about how to generate such applications. Some of these roles are new, others are relatively unchanged from those in traditional software system development.

1. System Analyst – Specifies new systems in a domain (12:4). Responsible for developing the concept of operations (defining policy, strategy, and use of application) and defining training requirements (5).
2. System Engineer – Works with the system analyst to partition the system into subsystems and assigns the tasks to software or hardware development, as appropriate (2).
3. Domain Engineer – Possesses detailed knowledge about the domain and gathers all the information pertinent to solving problems in that domain (12:4). Models the real-world entities required to satisfy the policy, strategy, and use of an application as defined by the system analyst. Determines how, if possible, these entities can be modeled within the constraints specified by the software engineer (5).
4. Software Engineer – Designs new software systems in the domain (12:4). Responsible for defining a formalized structure for the domain knowledge and providing the translation from the domain-specific terms to executable software (5).
5. Application Specialist – Uses systems in the domain (12:4). Familiar with the overall domain and understands what the new application must do to meet the requirements (a sophisticated "user"). Provides the application-specific information needed to specify an application.

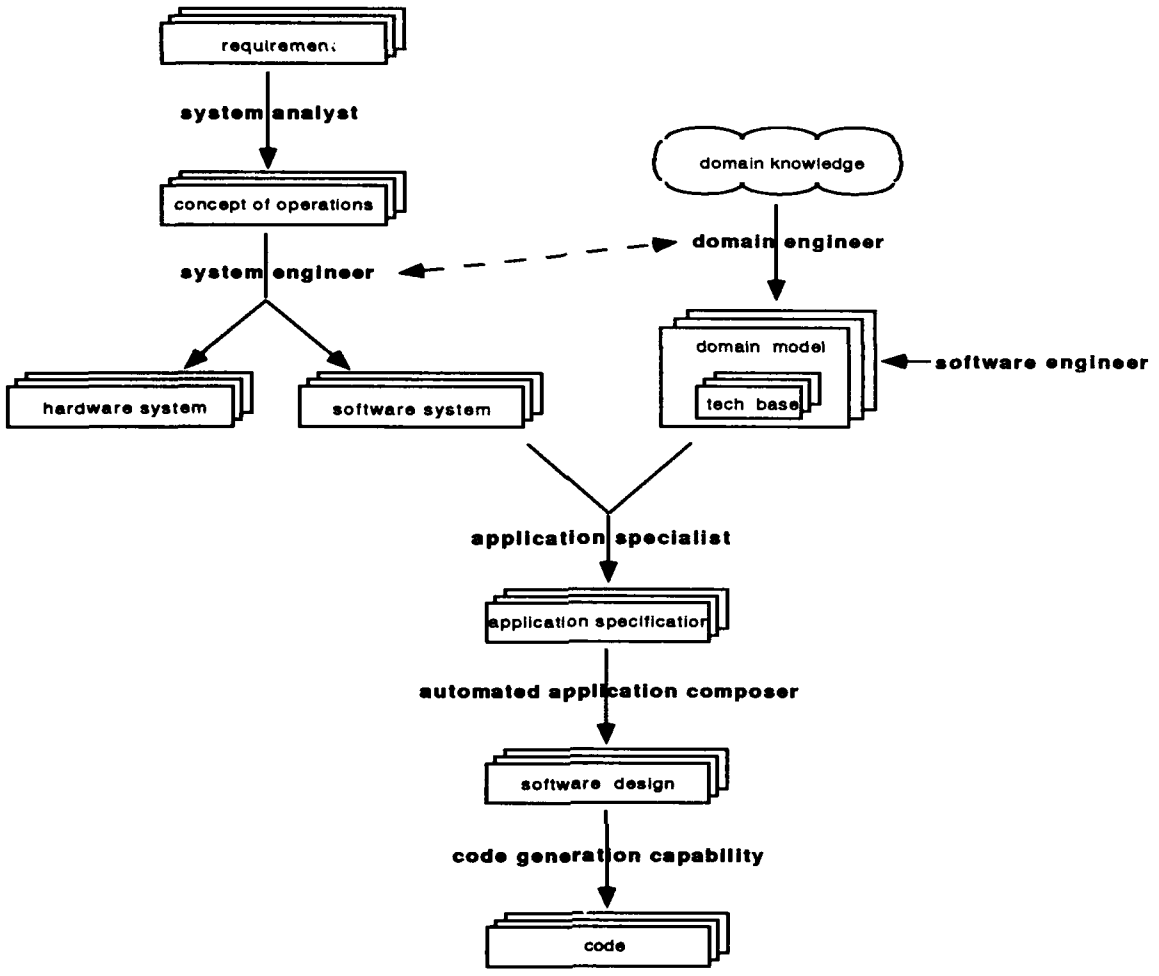

Figure 3.1. Roles

The relationships among these roles are shown in Figure 3.1. Usually, a new system begins with the identification of a new requirement. This requirement, if valid, is forwarded to a system analyst who develops a concept of operations. The system analyst works closely with the system engineer who partitions the system into software and hardware subsystems. The system engineer consults the appropriate domain engineer to define which components of his domain will be needed for software applications in the domain. The domain engineer and the software engineer decide on which components are needed to model the domain. The software engineer formalizes the domain knowledge provided by the domain engineer into a domain model and its technology base. The application specialist, using the domain model established by the software and domain engineers, cre-

ates a specification for an application. From this specification, an automated application composer generates a software design which is then input to a code generation capability.

3.3 General System Concept

3.3.1 Overview An overview of the application composition system's components and their relationships to each other appears in Figure 3.2. First, domain analysis is performed, which consists of gathering appropriate domain knowledge, formalizing it via a domain modeling language, and storing it in a domain model. The structure of the domain model is determined, in part, by the domain modeling language (DML) chosen. The software architecture model, like the DML, imposes a specific structure on the domain model, on the grammar used by the application specialist, and, ultimately, on the final application specification. The domain model is used to develop a domain-specific grammar. Although it may be transparent to the application specialist, he actually uses two grammars: one to identify domain-specific information and one to specify the architecture of the application. The architecture grammar remains the same for different domains; only the domain-specific grammar changes. Application-specific data is written using these two grammars and is converted into objects in the structured object base by the parser.

The populated structured object base and information from the technology base are combined to build an executable prototype. First, the application specialist performs semantic checking on the structured object base to ensure all constraints on the system have been met. He then executes the prototype to demonstrate the behavior of the proposed application. If the prototype does not behave as required, the application specialist can change the original input and re-parse it into the structured object base. Using the knowledge encoded in the domain model and the software architecture model, the structured object base is manipulated into a formal specification for a domain-specific software architecture (DSSA). The DSSA *is* the system design and becomes the basis from which code is generated. A visual system provides a graphical representation of the structured object base and the DSSA, as well as a means to add to or modify them.

The remainder of this section describes the above concepts and activities in more detail.

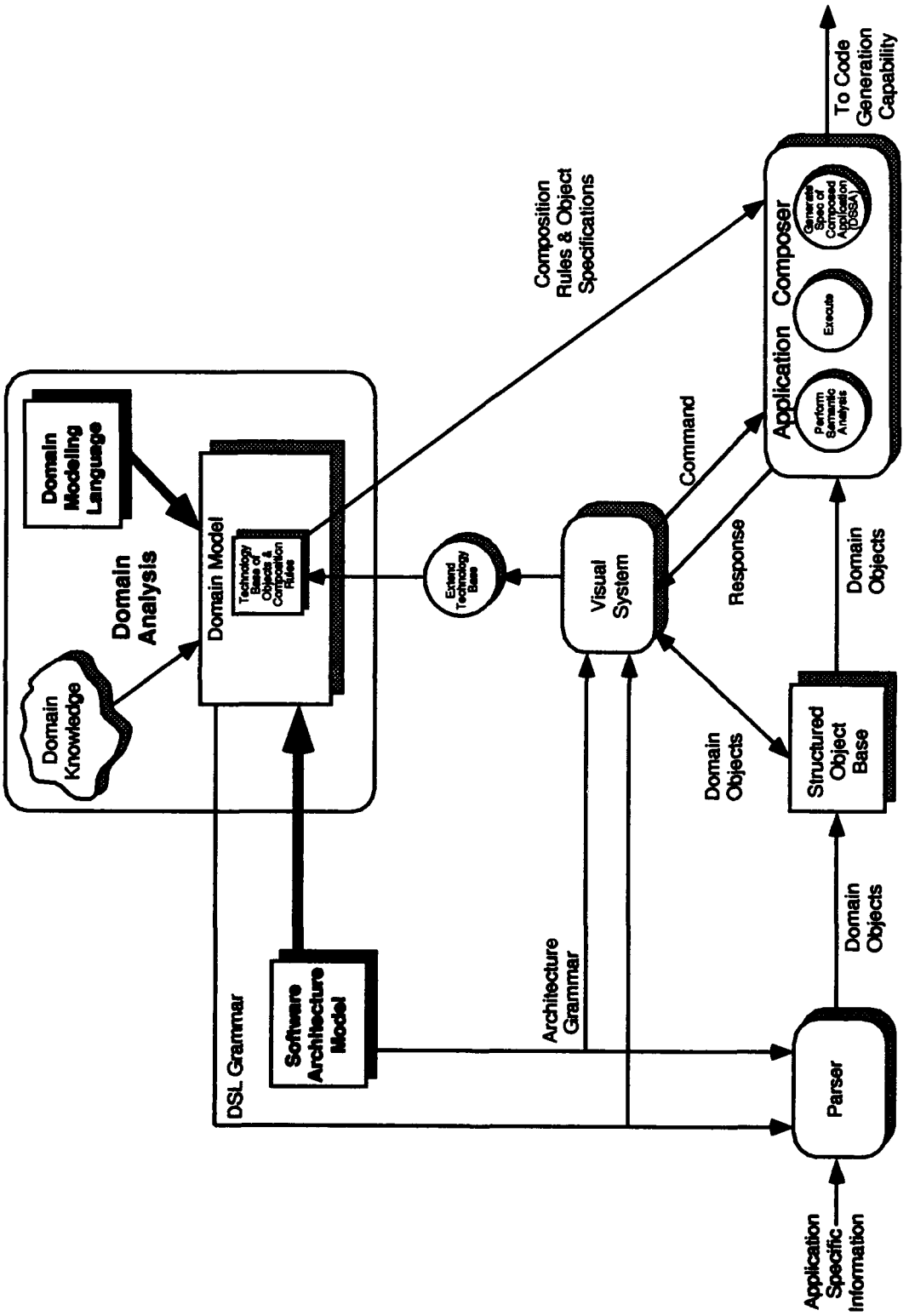

Figure 3.2. General System Overview

3.3.2 Developing a Formalized Domain Model Before any applications can be composed using this proposed system, the domain must be analyzed and modeled. In the software engineering context, a domain is commonly defined as "an application area, a field for which software systems are developed" (21:50) or "a set of current and future applications which share a set of common capabilities and data" (12:2). Identifying the boundaries of the domain, as well as "identifying, collecting, organizing, and representing the relevant information in a domain based on the study of existing systems and their development histories, knowledge captured from domain experts, underlying theory, and emerging technology within the domain" (12:2-3), constitutes domain analysis. Domain analysis is currently the subject of several other research efforts and is not directly addressed in this project. However, it is important to gather the basic data, formalize it, and store it in a standard format.

3.3.2.1 Domain Knowledge Domain knowledge is the "relevant knowledge" that results from a thorough domain analysis and later evolves naturally as more experience is gained solving problems in the domain (21:47). More specifically, domain knowledge consists of: basic facts and relationships, problem-solving heuristics, domain-specific data types, and descriptions of processes to apply the knowledge (3). In the context of this project, domain knowledge includes: descriptions of domain-specific objects (including their attributes and operations), data types, composition rules, and templates for commonly used architectural fragments.

3.3.2.2 Domain Modeling Language An analogy to a domain modeling language (DML) can be found in the more familiar data definition language of a database management system. A data definition language describes the logical structure and access methods of a database (14), just as our DML describes the logical structure of a domain model and defines how the objects can be accessed. A DML used to encode domain knowledge into a domain model must be able to formally describe:

1. Object Classes: Abstractions of real-world entities of interest in the domain.
2. Operations: Behavior of the objects in the domain.

3. Object Relationships and Constraints: Rules for relating objects (and sets of objects) to other objects, as well as the constraints on these relationships. Examples include:
   (a) Communication Structure: Message passing between/among domain classes and operations.
   (b) Composition Structure: Rules for combining domain object classes into higher-level application classes and operations into higher-level application operations.
4. Exception Handling: What to do when an error is encountered.

To be useful in an automated system, the domain knowledge must be encoded into a format that the software system can manipulate. This problem is analogous to encoding knowledge in an expert system, where human knowledge is gathered and represented as rules that allow a computer program to utilize the information. Neil Iscoe describes a method for encoding domain knowledge into a domain model (see (11) for details). He proposes using a domain modeling language or a meta-model as the basic framework to instantiate a domain model based on some operational goal(s) (reasons for which the knowledge will be used) (see Figure 3.3). Our operational goal is to "use the domain model, software architecture model, and structured object base to generate a software architecture for the application problem to be solved – to generate a domain-specific software architecture" (2).

3.3.2.3 Domain Model A domain model is a "specific representation of appropriate aspects of an application domain" (10:302) including functions, objects, data, and relationships (20). It is a result of expressing appropriate domain knowledge (identified by the domain engineer) in a domain modeling language with respect to certain operational goals (10:301-2).

Several researchers (4, 6, 7, 15) have indicated that software engineering must become more of an engineering discipline if we are ever to reap the benefits of design reuse (increased productivity, improved reliability, certifiability, etc.). When designing specific applications, engineers use models, "codified bodies of scientific knowledge and technology presented in (re)usable forms" (6:256) which are available to all practioners in various technology bases. Reuse of these validated, commonly-used models, which are readily available in

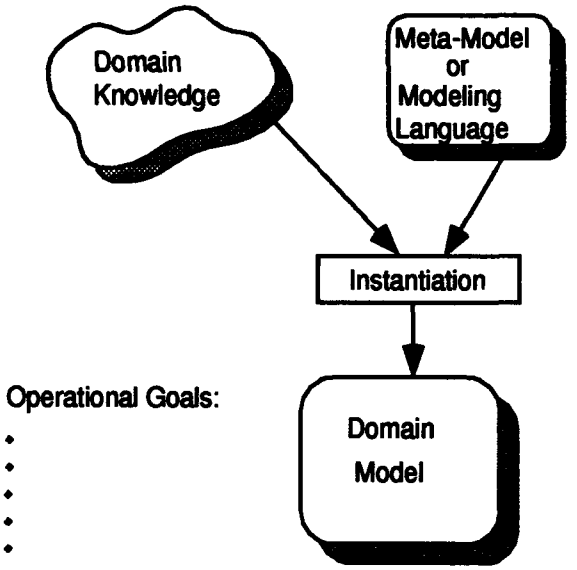

Figure 3.3. Domain Model Instantiation

various technology bases, allows the engineer to construct a practical, reliable solution to the problem at hand.

Contained within our domain model is such a technology base which acts as a repository for our reusable models. In our system, these models are often referred to as components. Using an object-based perspective, a component can represent a real-world entity, concept or abstraction and encompasses all descriptive and state information for that entity/concept/abstraction as well as its behavior (what operations or functions it performs and/or what transformations it undergoes). Components can be primitive domain objects as described above or a "packaging" of these objects whose structure is determined by the software architecture model. These packaged components will be referred to as architectural fragments since they can be used to build an application architecture. The technology base contains templates for generic components, rules for component composition, and descriptions of primitive object behavior. The parameters required to instantiate these generic templates will be specified by the application specialist.

Domain analysis reveals common features of the software architectures that can be used to implement various specific applications within the domain. In addition, common constraints are identified and codified into rules used to determine how software com-

ponents can be legally combined. Using rules allows additional flexibility; any specific architecture can be built as long as it meets the criteria specified by the rules.

3.3.3 Building A Structured Object Base Several steps must be taken to build the structured object base. The following system components are essential to this phase.

3.3.3.1 Domain-Specific Language As with our domain modeling language, an analogy to a domain-specific language (DSL) can be found in a data manipulation language from the realm of database management systems. In the database context, a data manipulation language allows the user of a database to retrieve, insert, delete, and modify data stored in the database (14:13). In our context, a DSL is a language with syntax and semantics which represents all valid objects and operations in a particular domain, allowing modeling and specification of systems within that domain (22). According to James Neighbors, a domain language is a machine-processable language derived from a domain model. It is used to define components and to describe programs in each different problem area (i.e., domain). The objects and operations represent analysis information about a problem domain (18). In our research, a domain-specific language is defined as a formal language used to define instances of objects and operations specific to a domain.

The objective of our DSL is to generate the structured object base needed to specify an application architecture within a specific domain. To do so, it must be able to:

1. Instantiate objects
2. Instantiate generic objects
3. Instantiate generic architectural fragments
4. Compose the instantiated objects and architectural fragments in some meaningful way

The object classes defined in the domain model are merely templates or patterns to be used when constructing objects; they do not refer to specific, individual objects. The first sentence type listed above creates specific instances of the objects in the object base. These objects are used in building architectural fragments or as parameters for generics. Default values can be used for attributes so these values need not be entered through the DSL every time they are used.

Generics, stored in the technology base, provide templates for commonly used objects and components; thus, the application specialist need not start from scratch each time he wants to include one of these commonly used components. Generics must be instantiated before they can be used. Instantiation is done by specifying which model is to be used and providing specific instances and/or other data, as required. For example, a generic architectural fragment may use three objects of a certain class. When this generic is instantiated, three specific object instances of the required class must be given.

3.3.3.2 Software Architecture Model In addition to identifying the objects to be used in generating a particular application, the application specialist must indicate what is to be done with those objects; i.e., he must identify the application operations. Domain primitive operations, associated with primitive objects, are available in the technology base. But how can these primitive operations be assembled (composed) into application-specific operations? What are the rules for composing these primitive operations into application operations? How can these rules be represented and implemented?

Software architectures provide insight into software system composition. In its most fundamental sense, an architecture is a recognizable style or method of design and construction. A software architecture has been defined as "a template for solving problems within an application domain" (28:2-2) or "the high level packaging structure of functions and data, their interfaces and controls, to support the implementation of applications in a domain" (12:3). It provides a mechanism for separating "the design of (domain) models from the design of the software" (5). This separation of domain knowledge from software engineering knowledge allows each type of engineer to concentrate on the issues relevant to his own area of experience, without becoming an expert in the other discipline. By focusing only on the design of the software, the software engineer is able to develop simplified packaging and control structures which can be reused across a wide variety of domains.

Because a software architecture serves as a structural framework for software development, we can expect it to provide a consistent representation of system components as well as the interfaces between those components. A standard representation ensures that each component is developed in the same manner, eliminating many implementation choices and

simplifying the development process. This standardization also results in consistent interfaces between all components, enabling them to be easily combined. This consistency of component representation and interfaces should provide a suitable and flexible framework for composing primitive operations into application-specific ones.

3.3.3.3 Architecture Grammar Certain portions of the application specialist's input are not dependent on any particular domain; rather, they depend on the software architecture model. These architectural aspects of the application can be specified using a grammar common to all domains, an architecture grammar. This grammar enforces the structure imposed by the software architecture model by defining valid sentences for packaging the primitive domain objects into architectural fragments to define an application architecture. These sentences will compose application operations using domain-specific components described by the domain-specific grammar and other application operations.

3.3.3.4 Parser After the application specialist specifies the application components using the domain-specific language and architecture language, the input must be parsed into objects in the structured object base. The parser generates specific object instances whose initial states are determined by the application specialist's input.

3.3.3.5 Structured Object Base The structured object base contains application specific information: specific instances of domain object classes with all appropriate attribute values for determining the object's state, as well as relationships for both domain objects and operations. The kinds of objects that might populate the object base and the overall structural framework of those objects (the shape of the abstract syntax trees) are established by the domain and software architecture models. The specific object instances and the actual structure of the object base are determined by the application-specific information provided by the application specialist using the DSL and architecture grammars.

3.3.4 Composing Applications The application composer generates the application architecture specified by the application specialist. This is accomplished by combining the appropriate instantiated domain objects from the structured object base in accordance

with the domain composition rules. After the architecture is generated, its behavior can be simulated to demonstrate its suitability and correctness. It should be noted that the operations associated with each object in the technology base are certifiably correct; that is, individual objects are guaranteed to behave as required. However, the specific objects which are composed into the application may have been combined in such a way that the composed application may not behave as expected or required. When the application specialist is satisfied that the composed architecture is actually the one desired, he can generate a formal specification for the architecture which can later be used to develop a fully coded system.

3.3.4.1 Semantic Analysis After an application is identified, the next step is to ensure that the specified composition is appropriate; i.e., that it makes sense and meets the constraints imposed by the composition rules. This step is accomplished via a semantic analysis phase. As in programming language compilers, one aspect of semantic analysis is to verify that a syntactically correct construct, which satisfies the restrictions of the grammar in which it was written, is "legal and meaningful" (8:10). To be legal and meaningful, the proposed application must meet certain other composition restrictions: e.g., components must already exist before they can be used, an input to one component must be produced as an output from another component, etc. Another aspect of semantic analysis is to use knowledge about domain objects and typical system constructions to assist the application specialist in choosing the components needed and in combining them appropriately to create applications which behave as desired. Errors identified during the semantic analysis phase must be corrected before the composition process can proceed.

3.3.4.2 Execute A composed application architecture that passes all semantic analysis checks is legal and meaningful, but does it do what the application specialist wants it to do? The execute component of the application composer simulates the behavior of the architecture, using object operations which specify each component's behavior. This behavior simulation may not be efficient or robust enough to serve as a full-scale operational system, but it provides the application specialist timely feedback on the correctness of the specified architecture. If the application is incorrect (i.e., it does not behave as re-

quired/expected), the application specialist reassesses the components which were used in the application and how they were combined, creating a new or editted application to satisfy his requirements. This ability to simulate execution behavior in this rapid-prototype manner assures the application specialist that the proposed application actually behaves correctly before a formal specification and fully-coded system are generated.

3.3.4.3 Generate Specification A legal, meaningful, and correctly composed application provides a software architecture which satisfies the application specialist's requirements for a particular application. The software architecture can be used as a blueprint, template, or specification from which to design and implement a full-scale, operational version of the application. The generated specification is intended to be in a formal, machine-processable format which can be used directly by a code generation tool to produce a fully-coded application. However, the specification format could be tailored to provide whatever form is appropriate for the using organization: graphical, textual, etc.

3.3.5 E~tend Technology Base Eventually, the technology base, which formalizes the knowledge about domain objects, will become outdated as understanding of the domain evolves and as the domain itself adapts to accommodate a changing technological environment. Although the technology base may appear to be static, it must be dynamic enough to accommodate this additional information as well as higher-level object classes and operations, generic components and architectural fragments that are developed. These additional elements give added flexibility to the application specialist because more predefined components are available for future applications

A specialized set of tools allows the technology base to be modified or extended to include this additional or revised domain knowledge. The extender must enforce the structure dictated by the domain modeling language and the software architecture model.

3.3.6 Visualization "A picture is worth a thousand words." This old adage is still true today, especially when dealing with complex and abstract concepts. The visual system provides the application specialist with a graphical view of the structured object base, as well as the application software architecture generated to satisfy his requirements.

By reviewing these "pictures," the application specialist can more fully understand the components available for composition and the application just composed. Moreover, the visual system will also be capable of inserting new instances of domain objects into the structured object base, editing domain objects already in the object base, and executing the application composer. It also provides the capability to extend the technology base, enabling the application specialist and/or the software engineer to add/modify domain object classes, add/modify generic components, and add/modify architectural fragments.

The visual system is addressed in more detail in (29).

3.4 Related Research

Several other research efforts have addressed various aspects of the system we are attempting to develop. This section summarizes this related work and analyzes the similarities to and differences from our project.

3.4.1 Hierarchical Software Systems With Reusable Components Don Batory and Sean O'Malley are working to incorporate an engineering culture into software engineering. The traditional engineering mindset dictates that new systems are created by fitting well-tested, well-defined, and readily available building blocks into a well-understood blueprint or architecture, which, if properly used, is guaranteed to produce the desired system. To this end, they have developed a "domain-independent model of hierarchical software design and construction that is based on interchangeable software components and large-scale reuse" (4:2).

In Batory and O'Malley's view, each interchangeable component consists of an interface (everything externally visible) and an implementation (everything else). Different components with the same interface belong to a realm. All the components in a realm are considered to be interchangeable or "plug-compatible" (4:3) because they have identical interfaces. Symmetric components have at least one parameter from their own realm and can be combined in "virtually arbitrary ways" (4:2) (also see Figure 3.4). Conceptually, components are seen as layers or building blocks for an application; a system is seen as a

stacking of components, i.e., a composition of components. Constraints on stacking components (i.e., rules of composition) are derived from the compatibility of their interfaces.

Hierarchical software system design recognizes that constructing large software systems is a matter of addressing only two issues: which components should be used in a construction and how those components are to be combined together (4:16). It employs an open software architecture, which is limited only by the inherent ability of the components to be combined, i.e., by their interfaces. Symmetric components have no inherent composition restrictions; thus, composition rules are simplified while ensuring maximum design flexibility and potential reusability of components.

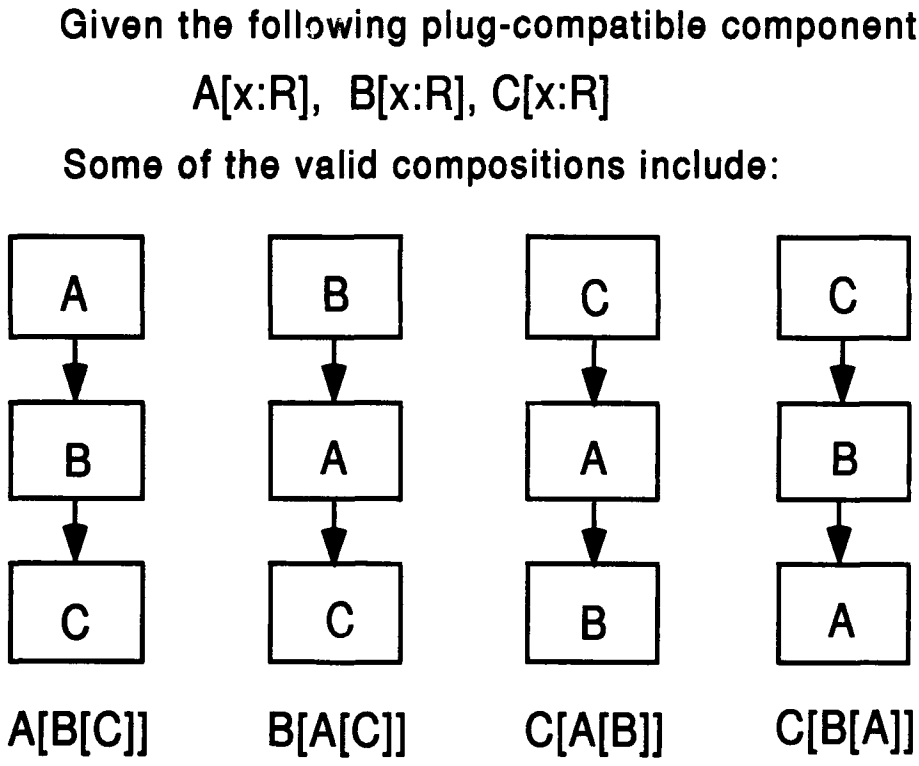

Figure 3.4. Combining Plug-Compatible Components

Batory and O'Malley use an interesting analogy, equating their concepts to a grammar, as shown in Table 3.1 (4:5). Using this analogy, a domain is a language. Consider the following example (4:5):

S = {a, b, c}	$S \rightarrow a \mid b \mid c$
R = { g[x:S], h[x:S], i[y:R]}	$R \rightarrow gS \mid hS \mid iR$

A realm S, having a set of components (a, b, and c), corresponds to a production where the non-terminal S can be replaced by either a, b, or c. Whenever a component from realm S is needed, a, b, or c could be used, depending on the behavior and level of detail needed. A realm R, whose components g, h, and i require parameters from realms S, S, and R, respectively, can be represented by a production where a non-terminal can be replaced by both a terminal and a non-terminal. The non-terminals on the right-hand side are the realms from which the parameters are provided. The complete analogy is summarized in Table 3.1.

Concept	Grammar
Parameterized Components	Productions with non-terminals on right
Parameterless Components	Productions that only reference terminals
Symmetric Components	Recursive production
Component Interface	Left side of a production
Implementation	Right side of a production
Realm	Set of all productions with the same head
Software System	Sentence
Rules of Composition	Semantic error checking

Table 3.1. Analogy to Grammar

Batory and O'Malley's work provides support for our research. It confirms the underlying principle of an application generator: building software systems from reusable components is "simply" a matter of selecting which components to use and deciding how to compose them together. It reinforces our intention to use an object-oriented approach in designing our system. It also illustrates the role of component interfaces in system composition and demonstrates the importance of consistent interfaces and composition styles in developing rules for combining components.

On the other hand, the Batory/O'Malley work falls short, in some ways, of what we are attempting. It does not incorporate a mechanism for an application specialist to specify new applications in domain-specific terms; this is a primary emphasis of our project. It also does not seem to provide for tailoring of component composition to suit the application being built; composing component A with component B into component C will always produce the same behavior for C. We want to be more flexible in our compositions and allow A and B to be composed into C in one situation and C' in a different situation, depending on how the application specialist specifies the composition.

3.4.2 Automatic Programming Technologies for Avionics Software The Lockheed Software Technology Center has developed the Automatic Programming Technologies for Avionics Software (APTAS) system pictured in Figure 3.5 (16:2). The APTAS system, built for the target tracking domain, "takes a tracking system specification input via user interface with dynamic forms and a graphical editor, and synthesizes an executable tracker design" (16:1). An application specialist defines a new tracking application by answering questions which appear in pop-up, menu-like forms. His answers determine which additional questions are to be asked as he is guided through specifying a new tracker. When all pertinent specifications have been entered (defaults exist for questions which are left unanswered), the application specialist generates a software architecture for the new tracker via the architecture generator. A graphical user interface provides a "picture" of the application architecture and allows the user to change it interactively. After the application specialist is satisfied with the architecture just created, he generates executable code to implement that architecture via the synthesis engine (16). He can also invoke a run-time display which facilitates testing and analyzing the tracker just created.

The Tracking Taxonomy and Coding Design Knowledge Base is at the center of the APTAS system. It contains the system's specification forms, the primitive modules from which new trackers are constructed, and the composition rules which establish how primitive modules are to be combined. The application specialist's answers to the questions on the specification forms progressively reduce the number of primitive modules which are candidates for incorporation into the new tracker. The architecture generated upon

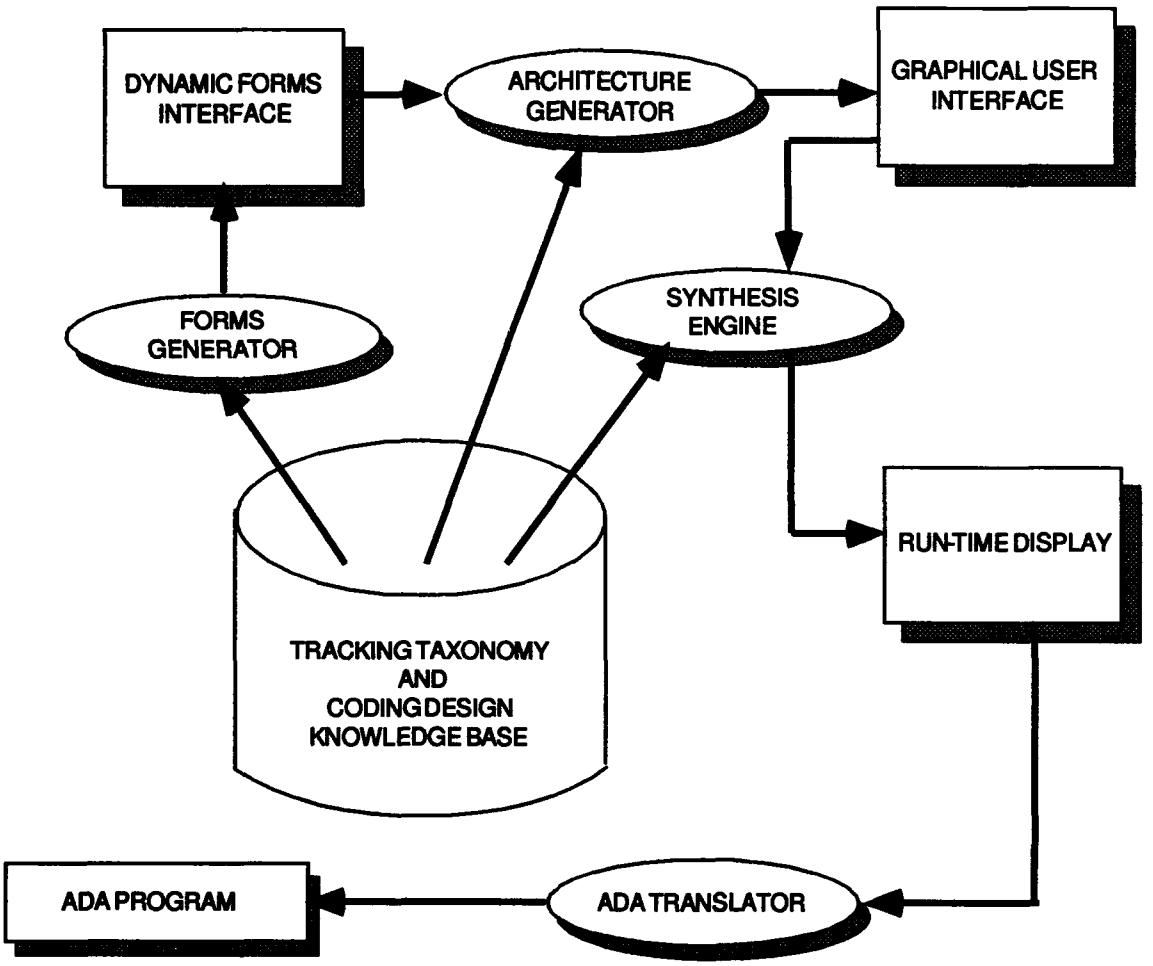

Figure 3.5. APTAS

completion of the forms specification is synthesized into an executable intermediate language, Common Intermediate Design Language (CIDL). The CIDL code can be executed to demonstrate system behavior. If the system behaves as desired, the CIDL representation can then be transformed into Ada code. The use of an intermediate representation, such as CIDL, localizes the code translation function and enables languages other than Ada to be targeted more easily.

The APTAS primitive modules and their composition rules are also written in CIDL. Extending the system involves writing new primitive modules and incorporating references to these new modules into the appropriate composition rules and specification forms. This is generally considered to be a software engineer's task (rather than an application spe-

cialist's), as CIDL is a software specification language and few tools exist to simplify the process.

APTAS is strikingly similar to the system we envision. It clearly demonstrates that the concept of user-initiated composition and generation of domain-specific systems is feasible. It allows application specialists to specify new applications in domain-specific terms, by way of menu-like specification forms. It also provides a sophisticated graphical user interface which can be used to construct and/or edit the tracker system, as well as to view the structure of the architecture.

There are, however, some major differences between APTAS and the system we are developing. APTAS's use of a domain-specific language is implicit and embodied in its graphical user interface. Our domain-specific language, on the other hand, is explicit and its grammar is usable in both textual and graphical modes. We believe this provides advantages to both the software engineer and application specialist in terms of adaptability, flexibility, and ease of use. In addition, APTAS currently lacks a set of convenient tools to facilitate extending its knowledge base; such a toolset is an integral part of our system.

3.4.3 Model-Based Software Development The Software Engineering Institute's (SEI) Software Architectures Engineering (SAE) Project has proposed a concept called Model-Based Software Development (MBSD) (15). Like Batory and O'Malley, MBSD strives to apply traditional engineering principles to software development by exploiting prior experience to solve similar problems. This prior experience is codified in models, "scalable units of reusable engineering experience" (15:11), which are stored in a technology base. In a mature engineering domain, the technology base will contain "all the components an engineer needs to predictably solve a class of problems, and the tools and methods needed to predictably fabricate a product from the components specified by the engineer" (15:4). Under MBSD, software development follows the engineering paradigm: reuse existing, mature models rather than starting from scratch for each new development. This involves much more than code reuse; the requirements analysis, design, and software architecture are reused each time the corresponding model is used.

MBSD uses a technology base, a repository of models and composition rules that share common engineering goals. Each model is mapped to a specification form and a software template for the target application language. The specification form is a text-based description which uniquely identifies a specific instance of a model. The software template is code containing place holders, which are replaced with information from the specification form (15:10).

As part of MBSD, the SEI uses the Object-Connection-Update (OCU) model as a consistent pattern of design, a software architecture. This model is especially suited to domains where the real world can be modeled as a collection of related systems and subsystems (15:17). Partitioning a system into subsystems provides different levels of abstraction, giving the flexibility to replace a subsystem with another that either provides a different function or has a different level of detail. In the OCU model, subsystems consist of a controller, a set of objects, an import area, and an export area as pictured in Figure 3.6 (15:18).

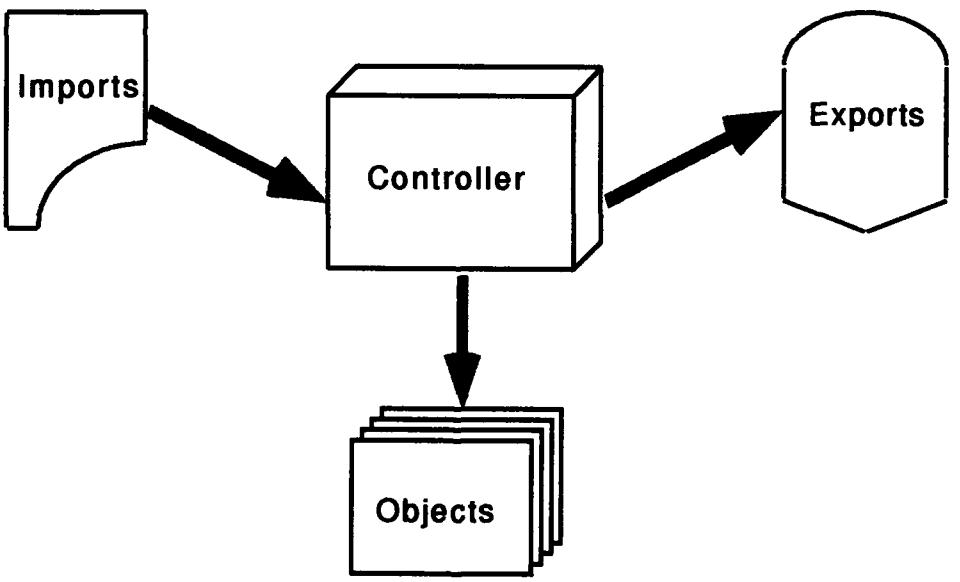

Figure 3.6. OCU Subsystem Construction

1. Controller – Performs the mission of the subsystem by requesting operations from the objects it connects. A controller is passive, triggered by a call to perform its mission, and depends on the other subsystem components to accomplish that mission.

2. Objects – Model behavior of real-world entities and maintain individual state information. An object is passive, triggered by a call from the controller to which it is connected.
3. Import Area – Makes data external to the subsystem available to the controller and its objects.
4. Export Area – Makes data internal to the subsystem available to the other subsystems.

Both controllers and objects have standard procedural interfaces used by external controllers or application executives to invoke some action. Controllers have the following procedures (15:19):

1. Update – Updates the OCU network based on state data in the import area and furnishes new state data to the export area.
2. Stabilize – Puts the system in a state consistent with the current scenario.
3. Initialize – Loads the configuration, creates objects, and defines the OCU network.
4. Configure – Establishes the physical connection between import area and input data as well as export area and the output data.
5. Destroy – Deallocates the subsystem.

All objects have procedures analogous to those for controllers, but operating on a single object instance. Specifically, these procedures are (15:20):

1. Update – Calculates the new state based on input data and the current state.
2. Create – Creates a new instance of the object.
3. SetFunction – Changes or redefines the function used to calculate the state.
4. SetState – Directly changes the object's state.
5. Destroy – Deallocates the object.

These well-defined and consistent interfaces for controllers and objects facilitate and simplify the application composition process.

MBSD provides some significant insights upon which to base our research effort. Its focus on the reuse of validated, engineering experience is attractive and we have adopted the notion of storing such information in a technology base. The OCU model provides a realistic approach toward composing primitive objects into application-specific subsystems.

3.4.4 Extensible Domain Models[2] The Kestrel Interactive Development System (KIDS) is a knowledge-based system that allows for the capture and development of domain knowledge (27). The representation of the domain knowledge constitutes a domain model, and these domain models are called domain theories. Essentially, the domain theory provides a formal language, natural to specialists in that domain, for specifying the problem they want to solve. The KIDS system provides support for constructing, extending, and composing domain theories, and over 90 theories have been built up in the system (27). Additionally, the set of domain theories developed during the domain modeling effort serves as the basis for software synthesis.

The foundations of the KIDS approach emerged from years of research into the specification and synthesis of programs (27). Concepts from algebra and mathematical logic are used to model application domains and synthesize verifiably correct software. Domain modeling entails the analysis of the domain into the basic types of objects, the operations on them, and their properties and relationships. The domain model is then expressed as a domain theory. Theories are useful for modeling application domains for the following reasons.

1. The basic concepts, objects, activities, properties, and relationships of the domain are captured by the types, operations, and axioms of a theory.
2. Any queries, responses, situation descriptions, hypothetical scenarios, etc. are expressed in the language defined by the domain theory.
3. The semantics of the application domain are captured by the axioms, inference rules, and specialized inference procedures associated with the domain theory.
4. Simulation, query answering, analysis, verification of properties, and synthesis of code are supported by inference within the domain theory.
5. Various operations on models such as abstraction, composition, and interconnection are supported by well-known theory operations of parameterization, importation, interpretation between theories, and others. Thus, a high degree of extensibility is obtained.

[2]This section was provided by Major Paul D. Bailor

3.5 Specific System Concept

Several aspects of the system described in Section 3.3 depend heavily on the choice of the models and tools used in the implementation. These selections may impact other parts of the system. Figure 3.7 is a modification of the system overview, incorporating the specific models and tools to be used. It represents Architect, the specific system which is to be implemented during this research effort.

3.5.1 System Overview Figure 3.7 illustrates how specific tools and models further define Architect. REFINE, as the domain modeling language, imposes its structure on the domain model (which will be represented in REFINE also). Input, written in the domain-specific and architecture grammars, is processed through a parser generated by DIALECT. DIALECT requires two inputs to generate a parser: a DIALECT domain model (a subset of the system domain model) and a grammar definition. The DIALECT parser creates abstract syntax trees in the structured object base. The visualizer will be implemented using INTERVISTA. The SEI's OCU model will serve as our software architecture model, providing a structure around which to generate our applications. KIDS will serve as a mechanism for realizing extensibility of the domain model and technology base.

3.5.2 Software Refinery Software Refinery is a formal-based specification and programming environment developed by Kestrel Institute and available commercially from Reasoning Systems, Inc. We have selected this environment in which to implement Architect for several reasons, but the main factor in our decision is REFINE's powerful, integrated toolsets that allow rapid prototyping. This decision has many implications on how the system will operate, as we will show.

3.5.2.1 Capabilities The REFINE environment consists of the following tools:

1. A programming language (REFINE) which includes set theory, logic, transformation rules, pattern matching, and procedures (24:1-2). The REFINE language provides a wide range of constructs from very high level to low level, making it suitable for various programming styles, including use as an executable specification language.

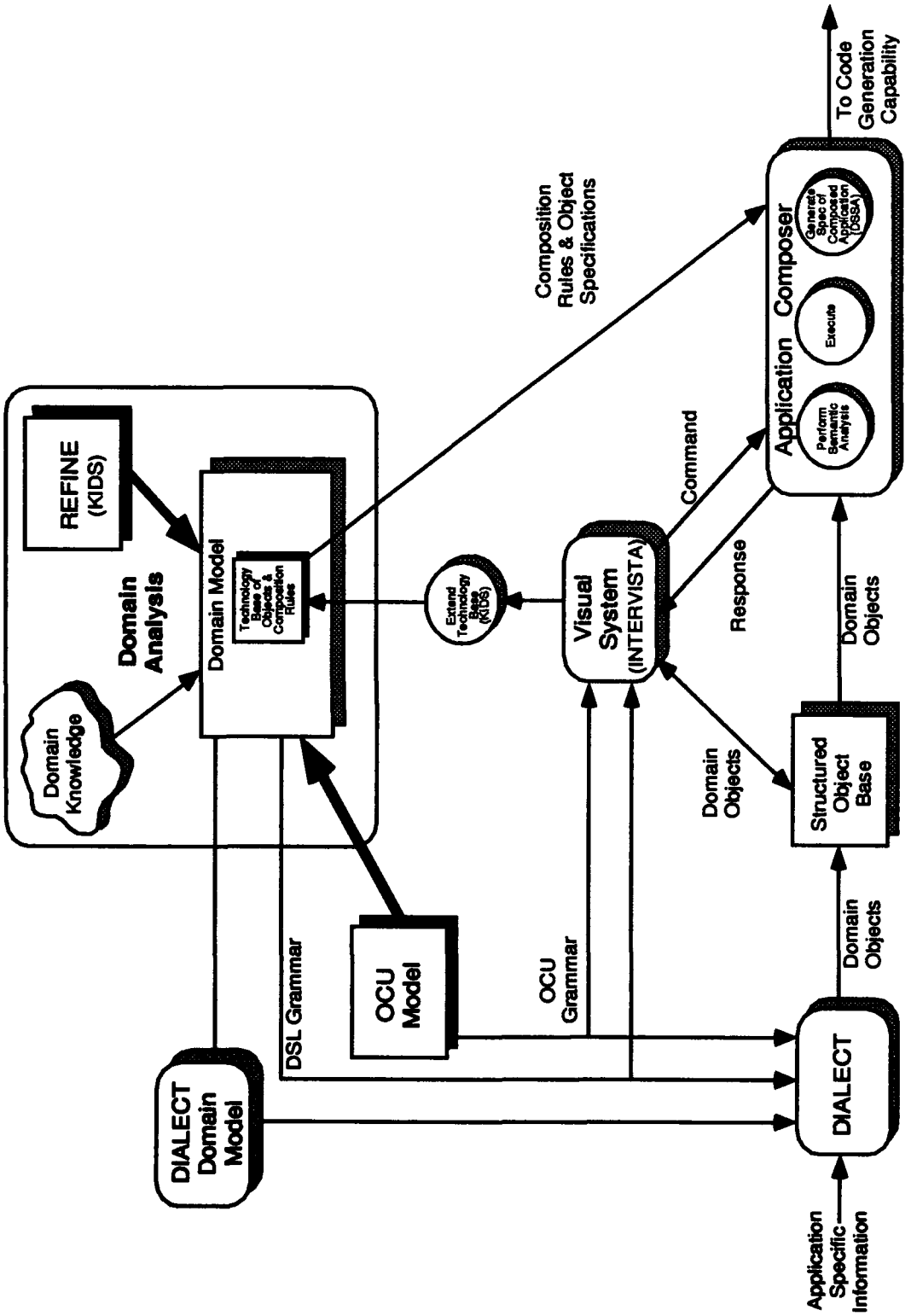

Figure 3.7. Overview of Specific System

2. An object base which can be queried and modified through REFINE programs (24:1-2). "Object classes, types, functions and grammars are among the objects you can define and manipulate" (24:1-4) with several built-in and powerful object base manipulation tools.
3. A language definition facility (DIALECT) which allows design of languages using an extended Backus Naur Form notation. REFINE supplies a lexical analyzer, parser, pattern matcher, pattern constructor, and prettyprinter for the language (24:1-2).
4. A toolset (INTERVISTA) which is useful in creating a visual, window-based interactive user interface.

3.5.2.2 Domain Modeling Language Some domain modeling languages already exist for expressing domain knowledge within a formalized domain model; we considered two such languages: the Requirements Modeling Language (RML) and REFINE.

RML was designed as a research tool as part of the Taxis Project at the University of Toronto. It allows "direct and natural modeling of the world" (9:3) in an object-oriented manner which "captures and formalizes information that is left informal or not documented in current approaches" (9:1). RML can express "assertions (what should be true in the world), as well as entities (the 'things' in the world) and actions (happenings that cause change in the world)" (9:4). This is precisely the type of information we want to capture in our domain model.

Even though both RML and REFINE appear to be capable of expressing the kind of information we require in the domain model, we chose REFINE as our domain modeling language for the following reasons:

1. REFINE provides an integrated environment including programming constructs and powerful object base manipulation tools. Use of REFINE's existing tools eliminated the need to write our own, allowing more time to be spent on the research itself.
2. RML is not an executable language; no compilers currently exist. To use RML, we would be forced to develop a compiler, a considerable overhead to our project. As REFINE is also capable of expressing the information we require, it is unclear what added benefits RML could provide to justify this additional expense.
3. The REFINE environment includes compatible tools (DIALECT and INTERVISTA) useful in other portions of the system.
4. REFINE is a commercially available and supported product.
5. Members of the research team already possessed a working knowledge of REFINE.

3.5.2.3 Parser "DIALECT is a tool for manipulating formal languages" (23:1-1). A part of the REFINE software development environment, DIALECT generates appropriate lexical analyzers, parsers and pretty-printers for user-specified, context-free grammars. Valid input is parsed and stored as abstract syntax trees in the REFINE object base, according to the structure established in the DIALECT domain model. The DIALECT domain model defines object classes, object attributes, and the structure of the instances in the object base. DIALECT also supports grammar inheritance, allowing for a base language with several variations or "dialects." In Architect, the architecture grammar acts as the common base, and the domain-specific grammar specifies a particular variation. DIALECT does impose restrictions on the grammars. Since DIALECT generates an LALR(1) parser, the grammar must be consistent with this type of parser. Also, the productions in the grammar must correspond to the structure defined in the domain model. Altering some productions may require updating the DIALECT domain model.

3.5.2.4 Structured Object Base The structured object base was implemented using the REFINE object base. REFINE includes many tools which, when combined with REFINE code, provide all of the functions necessary to manipulate the structured object base. However, the object base must be accessed through the REFINE environment.

3.5.2.5 Technology Base Models in the technology base were represented as REFINE code and stored in REFINE's object base. Although separate conceptually, the technology base and structured object base are not physically separate. Access is controlled by Architect to avoid any confusion.

3.5.2.6 Visual System INTERVISTA provides a tool set with which to generate a window-based graphical user interface. It is compatible with the other REFINE tools; therefore, it is easily integrated. INTERVISTA can access the REFINE object base, so all its required data is readily available.

3.5.3 Object-Connection-Update Model We have selected the Software Engineering Institute's Object-Connection-Update (OCU) model for our software architecture model. As such, it provides a framework for composing applications – a standardized pattern of

design for all applications and their components. The OCU model's consistent interfaces enable all components to be accessed in the same manner and its intercomponent communication scheme ensures that each component can readily access the external data needed for its processing. Currently, our focus is on implementing the subsystem aspect of the OCU model; the hardware interface portion of the model will be addressed in follow-on research efforts.

The choice of the OCU model for our software architecture model had certain implications for Architect.

1. Terminology – In keeping with the OCU model, we will refer to domain primitive objects as "objects," compositions of objects as "subsystems," the locus of control of a subsystem as a "controller," and the overall application itself as an "executive" (see (1) for a more detailed discussion of the executive). External data needed by an object are "input-data," whereas data to be made externally available are "output-data." An "import area" serves as a focal point for all external data needed by the subsystem and an "export area" is the focal point for all internal data to be made available to other subsystems. The OCU model's names for the object and controller procedural interfaces have also been retained.
2. Use of a Technology Base – Although the concept of storing reusable domain knowledge or models in a technology base is not unique to the OCU model, it is a fundamental component of Model-Based Software Development of which the OCU model is a part.
3. Domain Analysis – The OCU model deals with objects and subsystems. This imposes a constraint on the domain engineer and will impact the manner in which domain analysis is conducted. Under the OCU model, the domain engineer must model the domain in terms of subsystems which can be composed from lower-level, more primitive objects. Many domains can be naturally modeled in such a way; with other domains, a new mindset may be needed to incorporate the subsystem/object requirements of the OCU model. Alternatively, an additional class of software architectures may need to be defined.
4. Definition of Domain Objects – The OCU model requires that all objects be defined in the same manner. Each object has state data, other descriptive information, input-data/output-data definitions, and the following procedural interfaces: Update, Create, SetFunction, SetState, and Destroy. These requirements dictate how the objects will be constructed, severely limiting implementation choices. However, it is this very limitation which provides the flexibility that allows the domain objects to be successfully composed to satisfy the application specialist's specification.
5. Definition of Architectural Fragments – The OCU model requires that all architectural fragments (subsystems) be described in the same way. All subsystems have an import area, export area, controller, and objects. Each controller has the following procedural interfaces: Update, Stabilize, Initialize, Configure, and Destroy. As

with the objects, this apparent limitation on implementation choices actually provides great flexibility in composing subsystems and combining them into a complete application.

6. Composition Rules – The standardized object/subsystem definitions and interfaces of the OCU model simplify application composition. There are no inherent restrictions preventing one component from being combined with another; all composition rules are domain-specific and do not derive from the software architecture.
7. Intercomponent Communication – The OCU model establishes and enforces a standard method for intercomponent communication. Communication external to the subsystem is localized in the import area which obtains the necessary input-data for all objects within the subsystem. This localization of communication concerns within the narrow guidelines imposed by this scheme simplifies intermodule communication: subsystems can readily obtain needed external information in a consistent manner and changes in the low-level implementation of the communication process are hidden from the subsystems/objects.
8. Structure of the Resulting Application Specification – Obviously, the specification produced by the application composer is impacted by the choice of a software architecture model. The OCU model produces an application (an "executive") which is composed of subsystems. These subsystems can be decomposed into objects and lower-level subsystems, if appropriate. This hierarchical structure is preserved in the generated specification.

The OCU model is the result of years of research and experimentation by the SEI. It has been used successfully in the flight simulator, missile, and engineering simulator domains (5) and appears to provide a suitable structure for composing applications within our application composition system.

3.6 Conclusion

Software engineering may be on the brink of a new era, an era in which software engineers develop knowledge about generating software systems and application specialists actually create the software systems using familiar, domain-oriented terms. Our research, which builds on important work already accomplished by various researchers, is designed to demonstrate the feasibility of such an application composer.

IV. System Design

This chapter describes the design for populating and manipulating the structured object base as described in Chapter III, using the Software Refinery Environment. It begins with a high-level description of the design of the overall system and is followed by a detailed explanation of the areas related to this thesis.

4.1 System Design Overview[1]

4.1.1 Design Goals Throughout the design process, an attempt was made to optimize several fundamental goals. These goals include:

4.1.1.1 Domain Independence Since Architect must be applicable to any domain, it should not directly incorporate (i.e., "hardcode") knowledge about a specific domain or type of domain; the technology base is the proper, sole repository for such domain-specific information. If any domain knowledge were to be included in Architect, code changes would likely be required before it could be used with a new or modified domain. Obviously, this greatly limits the applicability and usability of the system.

4.1.1.2 Extensibility It would be very naive to assume that an initial domain analysis will reveal all possible knowledge about a particular domain and that the domain model, which formalizes this knowledge, will never change. In reality, the domain model will continue to evolve as existing knowledge is further refined and/or new domain information is added to the system. If this evolution cannot be achieved easily, Architect will quickly become obsolete.

4.1.1.3 Flexibility Because the concept of application composers is rather new, we do not yet know how application specialists and software engineers will best be able to use them. Architect, therefore, must be flexible enough to allow multiple methods for performing various tasks and a wide range of application specification options.

[1]This Section was co-written with Capt Cynthia Anderson and also appears in AFIT Technical Report AFIT/EN/TR-92-5 and (1).

4.1.1.4 Usability Application specialists, the primary users of this system, must have some degree of software programming knowledge, but they can not be expected to have the same degree of understanding as a software engineer. Therefore, it is important that the system not require detailed programming knowledge from its users. In many cases, the goals of usability and flexibility conflicted, so a balance had to be found.

4.1.2 Concept of Operations The steps which are followed when using this application generator are depicted in Figure 4.1 as labels on the flow arrows. The application specialist must first identify all objects to be used in the application specification and enter (parse) them into the structured object base using the domain-specific and architecture grammars. Some of these objects may require further information before they are fully defined (e.g., previously saved objects must be located and loaded, "holes" in generic object templates must be filled, etc.); the application specialist provides this additional information by completing the application definition. Although the application definition may be considered complete from the user's point of view, some data needed by the system may not yet be directly available; preprocessing the application specification automatically generates this essential information. When the application is fully defined, semantic checks are performed to identify any composition errors, which must be corrected before the application's behavior can be simulated. At any time, the application specification can be changed (edited), usually in response to a semantic error or to include additional data. If no semantic errors exist and no additions/changes have been made, the application's behavior can be simulated (executed). If the application behaves as the application specialist intends, he may generate a formal specification for the composed application which will be used by an automatic code generator to produce a fully realized application. At any point in this process, the application specialist may save selected components of the application definition (or the entire definition, if desired) to the technology base where they will be available for future use.

4.1.3 Software System Design The eight steps outlined above correspond directly to Architect's eight top-level modules as shown in Figure 4.2. The highest level module, the visual system, will eventually control each of the other modules as well as all user in-

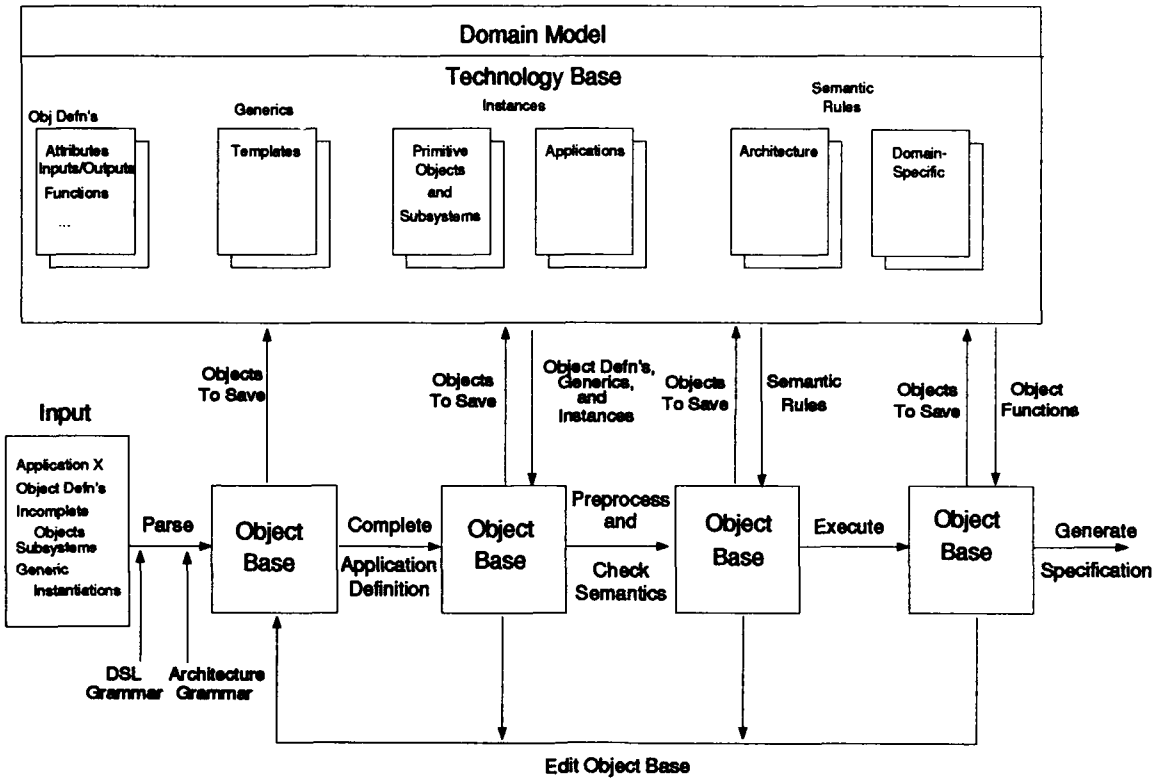

Figure 4.1. System Operations

teractions through a graphical interface. However, the basic system was developed before the visual system was completed; therefore, a simple user interface, which is easily replaced, was implemented. In the system design, we have made a conscious decision to keep application specification a domain-oriented, rather than programming-oriented, process. The system is designed to use all available domain knowledge to insulate the application specialist, as much as possible, from programming details, conventions, and jargon.

Each of Architect's major functions is encapsulated into one of the system's top-level modules. These modules are further discussed in the remainder of this section.

4.1.3.1 Parse Using REFINE, data can be input into the object base using one of two different methods: through a grammar or directly by using built-in REFINE functions.

Using the DIALECT tool allows the application specialist to reuse his input files as templates for other application definitions. The grammar also provides a consistent

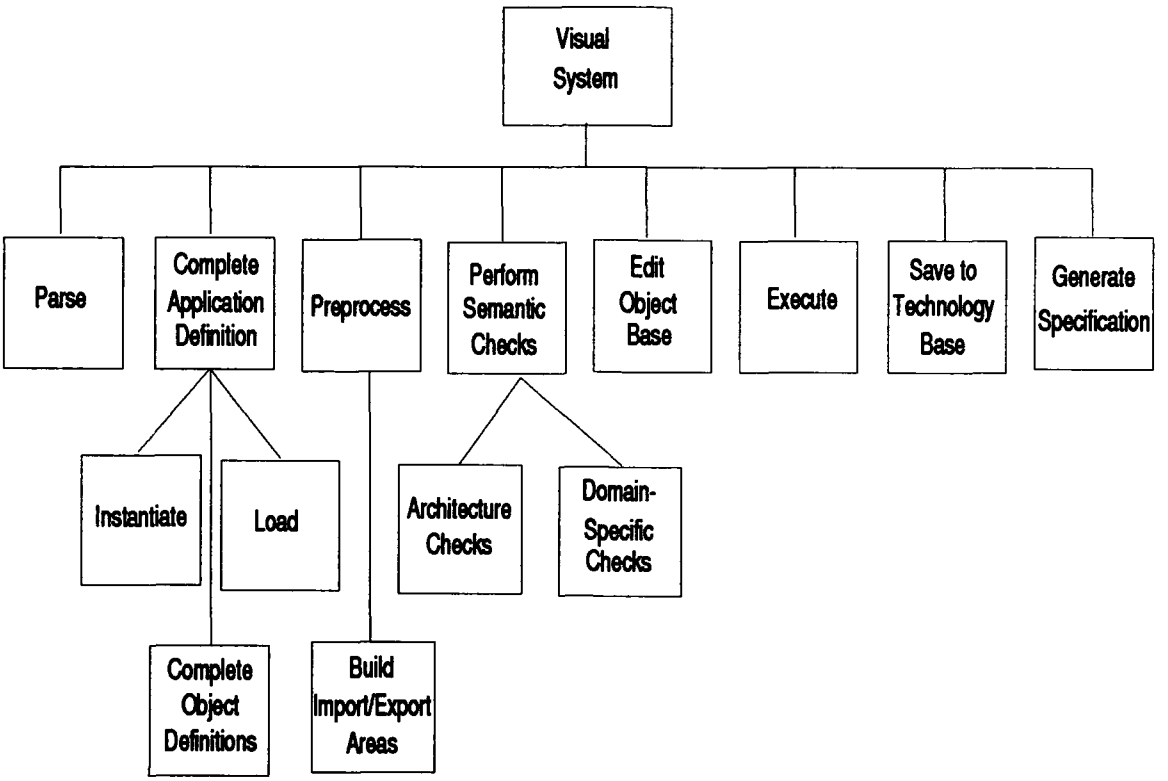

Figure 4.2. System Structure

format for saving objects from the object base to the technology base. The domain-specific portions of the grammar can be separated from the architectural components. DIALECT allows grammars to inherit the productions of another grammar. In this case, each domain-specific grammar inherits the same architecture grammar. If the domain is changed, only one grammar is affected. However, the application specialist must conform to the structure imposed by the grammars. If the current domain changes, the domain-specific grammar will require appropriate, corresponding changes. If a different domain is to be used, a new domain-specific grammar must be written; however, grammars for other domains can serve as a guide to facilitate creating new grammars.

An alternative approach is to build REFINE tools that allow the application specialist to interactively enter the objects into the object base. This method migrates most easily to the visual interface planned for a follow-on project (29). Also, this method is domain-independent. However, additional code must be written to save portions of the object

base. The developer must devise a standard format for the files to allow this data to be read back into the object base.

We chose to combine the two methods. The application specialist can input objects into the object base either through a grammar or interactively. The grammar provides a format for saving and retrieving all objects in the object base and a means of saving "templates" for application definitions. The interactive portions extend more readily to the visual system.

4.1.3.2 Complete Application Definition After all of the application specialist's input is parsed into the object base, additional processing is needed to complete the definition. The application specialist can fully define an object in the grammar or he can give partial information in one of three forms: a generic instance, an incomplete object, or an object to load. As part of completing the application definition, the system must actually instantiate the generic objects, complete incomplete objects by prompting the application specialist for values for each attribute, and physically load objects into the object base.

4.1.3.3 Preprocess Application The structured object base now contains only "complete", fully-instantiated application components. However, some critical data has not been specified. For example, the contents of a subsystem's import and export areas have not yet been identified. These areas are dependent upon the inputs and outputs, respectively, of the primitive objects which are controlled by that subsystem. Appropriate input-data and output-data for each primitive object have been identified during domain analysis and are available to the system in the technology base. Using this knowledge, the preprocessing module dynamically builds each subsystem's import and export areas, prompting the application specialist to indicate where the import data will be obtained when more than one subsystem produces the desired information.

4.1.3.4 Perform Semantic Checks Two levels of semantic checks exist in Architect. Architecture-oriented semantic checks ensure that the proposed application specification conforms to the composition requirements of the OCU model and that its behav-

ior can be successfully simulated (e.g., all components exist in the object base, application/subsystem update procedures directly reference only components which are part of the same application/subsystem, data input to one subsystem is produced as output by some subsystem in the application, etc.). Domain-specific semantic checks are knowledge-based, building on what is known about the domain, its objects, and previous applications created in that domain, to assist the application specialist in composing a meaningful and optimized application.

Meaningful architecture semantic checks can be performed on the application as a whole and also on its constituent subsystems. There are currently no meaningful semantic checks for primitive objects; the system assumes that primitive object class definitions and update procedures have been correctly constructed by a software engineer.

4.1.3.5 Edit Application If the application specialist decides an object instance is not exactly what was intended, he can edit the object. He can edit existing instances, add new objects, or delete objects. If the application specialist modifies the object base, he must also perform preprocessing and semantic checking on the entire object base to ensure the integrity of the data before simulating behavior or generating the specification.

The goal of domain independence has a large impact on how this module is designed. If certain domain knowledge is embedded in the source code, the code must be modified when the domain changes. If the code is completely independent, the interface may be more difficult to build and less user-friendly (the system can not give detailed prompts explaining what type of data is expected). In this case, domain independence is more important than friendly prompts.

4.1.3.6 Execute Application After the structured object base is fully populated and the semantic checks have uncovered no errors, the application's behavior can be simulated. This enables the application specialist to ascertain if the application, as specified, behaves as expected/desired. Behavior simulation is achieved by executing the application's **update procedure**, which consists of a series of calls to subordinate sub-

systems to execute their missions. Calling a subsystem to execute its mission invokes its **update procedure**. Subsystem update procedures consist of calls to subordinate subsystems/primitive objects, as well as **if** and **while** statements which allow conditional and iterative flows of control.

4.1.3.7 Save to Technology Base Since saved objects can be retrieved and parsed back into the object base, a function must exist to store objects in the object base into a file. The objects must be stored in the same format required by the input; that is, the format must adhere to the specifications of the domain-specific and architecture grammars. Saved application definitions can later be retrieved and loaded into the object base. Objects can be retrieved either through the grammar or through the interactive interface.

4.1.3.8 Generate Specification When the application specialist is satisfied that the application he specified behaves as desired, he can generate its formal specification. The formal specification provides all the information necessary to directly code the application into an efficient production system. Indeed, the formal specification generated by this application composer is intended to be input to an automated code generation facility.

4.2 Detailed Design

The previous sections described the overall design for our research. The details of the design pertinent to this specific thesis now follow.

4.3 Parse

4.3.1 Methods of Populating the Object Base REFINE users have two options to enter input into the object base: using a grammar or entering the data interactively. These two methods correspond to the two ways of populating the structured object base for Architect.

4.3.1.1 Grammar The DIALECT tool allows developers to produce domain-specific languages based on a DIALECT domain model. This domain model determines the structure of the abstract syntax trees by defining a hierarchy of object classes and their attributes. This information is also part of a system domain model because it describes the basic structure of domain-specific objects. Not all of the input is specific to a domain; parts will depend on the architecture model, which only changes when the architecture model changes. Therefore, two separate grammars can be used: one to describe the domain-specific objects, another to describe the architecture model objects. The domain-specific grammar requires changes when the domain changes, but the architecture grammar only requires changes when the architecture model changes. DIALECT supports grammar inheritance so each domain-specific grammar can inherit all of the productions of the architecture grammar. When defining a new domain, the software engineer generates a new DSL that inherits the architecture grammar.

Using a grammar, the application specialist can store the input for an application and only has to re-parse the file to populate the object base for that application. He can also use a previous application as a framework for a new one by modifying the file. In this case, applications do not have to be saved explicitly by the system; all the information is captured in the input file. There are disadvantages to this method. The major drawback is the fact that portions of the grammar are domain-specific and require changes when the domain changes. A grammar also imposes a structure on the application specialist; attribute values for each object class must be given in a specific order and each sentence must exactly follow the pattern dictated by the grammar. Using only a grammar to manipulate the object base, the application specialist cannot make simple changes to the object base; instead, he must correct the input file and re-parse it into the object base, unless additional functions are written for the system.

A visual system can provide a graphical, interactive interface which will solve some of the problems associated with having a highly structured input. If the visual system can be built using the grammar, the only real disadvantage to using a grammar is the fact that it must be changed whenever the domain changes. This cannot be avoided since the system must be domain-specific. One of the major challenges is building a visual system

that can use this grammar. The visual system uses a different type of interface to enter information than that required when parsing input through a grammar. It must be able to use the domain information embedded in the grammars and domain model so that the visual system itself remains a domain-independent implementation.

4.3.1.2 Interactive Input Adding objects interactively allows easy migration to a visual system since it provides an interactive interface. With this type of interface, the application specialist can be prompted for the values of each attribute when creating an object rather than trying to remember a structured format required by a grammar. Implemented correctly, the same code can be used for all domains because the code will be independent of the domain. The user only needs to specify what type of object to build and Architect can figure out the attributes needed and their data types. Changing to a new domain is simple; it only requires changing the domain model. However, there are disadvantages to using this method. It assumes that values for all attributes will be entered by the application specialist; i.e., there are no "internal" attributes that the application specialist should not modify. If this is not the case, Architect needs some method to determine which attributes are internal to the system and which must be entered by the application specialist. This could be done by storing a list of attributes that must be entered by the application specialist in the domain model.

Another disadvantage is with storing information in the object base. The format for each object type is defined by the object class of which it is an instance, but Architect has no overall format for storing aggregations of objects. A grammar already requires this structured format. Therefore, if a grammar is not used, an additional capability, including a format for the data, is needed to save and retrieve information that has been entered interactively. The application specialist can only save specific instances from the object base and cannot save templates for an entire application without adding another format specification.

4.3.1.3 Combination Each of the two methods has limitations that can be overcome by the other. By combining the two techniques the benefits of each can be realized. Portions of the input can be stored in a file and parsed into the object base

while the rest can be entered interactively. This method takes advantage of the benefits of each technique. The application specialist can save "patterns" for applications. He can either modify the file and parse in a new application or he can use incomplete definitions and give the additional data needed interactively. Either way, the same template file can be used for different applications. Also, objects and entire application definitions can be saved using the format determined by the grammars.

4.3.2 Creating Objects Regardless of whether the application specialist inputs the data in a file and parses it into the object base or enters it interactively, the user has several basic options when building objects in the object base:

1. Add a new primitive object instance (see Section 4.3.2.1)
2. Instantiate a generic primitive object (see Section 4.3.2.2)
3. Load an existing object from the technology base into the object base (see Section 4.3.2.3)
4. Add a new subsystem (see Section 4.3.2.1)
5. Instantiate a generic subsystem (see Section 4.3.2.2)
6. Load an existing subsystem from the technology base into the object base (see Section 4.3.2.3)
7. Parse an application from the technology base into the object base (see Section 4.3.2.4)
8. Build a new subsystem with assistance (using a predetermined pattern of control) (see Section 4.3.2.5)

4.3.2.1 Adding New Instances Architect allows the user to add new instances of primitive objects and subsystems using one of three methods:

1. Completely describe the instance through the grammar. If the user knows all the attributes values, he can specify them in the input file, and the parser will build an instance of the object with the given attribute values in the object base.
2. Indicate the type of object to be built in the grammar and add specific values interactively. The application specialist gives the object class name and the name for the specific instance. For example, if he is building a subsystem, but either doesn't know all of the details yet or doesn't want to include them in the file, he enters "object subsystem-obj, subsystem-name" where subsystem-obj is the name of the object class and subsystem-name is the name of the instance of the new subsystem. When the system completes the application definition, it will find all these partial definitions, build the corresponding object, and prompt for attribute values. The final result will

be a new object instance in the object base identical to objects parsed completely through the grammar.

3. Enter the object interactively. This allows the application specialist to enter an object after all of the input has been parsed. Architect will prompt for the object type and all the appropriate attributes, and create a new object instance in the application definition.

4.3.2.2 Instantiating Generic Objects Instantiating generic primitive objects and generic subsystems is done identically; Architect does not use any specific information about the object classes to instantiate a generic object. In fact, it cannot, because the code must be as independent of the domain as possible. The application specialist builds a generic instance that references a generic object and lists all the parameters. This generic instance is then instantiated as described in section 4.4.1

Generic instances entered through the grammar cannot be checked for the correct number and type of parameters while being parsed. The only way for the grammar to check the parameters is to build a different production for each generic object. Obviously, this makes it difficult for the software engineer to add or modify a generic since he not only has to build or edit the generic, he must modify the grammar and the DIALECT domain model as well. Since the generic object stores the types of the generic parameters, if the generic is instantiated interactively, the system can check that the user enters the correct number and type of parameters. This provides a more "user-friendly" method of entering generic instances. If a generic instance is entered incorrectly through the grammar, the system will give the application specialist the opportunity to edit the object.

An alternative approach is to store the generics as functions whose parameters are the generic parameters. The function builds the subsystem object based on the parameters and the "pattern" built into the function. With this method, it is more difficult to add new generics.

4.3.2.3 Loading Instances into the Object Base It is possible that specific instances of primitive objects or subsystems may be suitable for several applications. Rather than recreating these instances each time, they can be saved and later retrieved and reused. The purpose of loading existing domain primitive objects or subsystems is to parse one of

these instances into the object base for the current application. From the application specialist's view, these instances are a special case of generic primitive objects or subsystems that have no parameters. The major difference is that the exact copy of what is stored is parsed into the object base; none of the attributes change, including the name of the object. All of the domain-specific object classes and subsystems require unique names; therefore, at most one copy of each saved instance can be in the object base at any given time. If the instance needs to be used multiple times in a single application, it must be converted to a generic object by the software engineer. Objects can be loaded either through the grammars or interactively after the initial application definition has been loaded.

4.3.2.4 Loading Application Definitions into the Object Base Since entire application definitions can be saved, the application specialist must be able to retrieve these definitions from the technology base. When parsing an application into the object base, the application specialist specifies the name of the application (which corresponds to the name of the file) and Architect will parse all the objects associated with that application into the object base. If an application definition already exists in the object base, Architect will warn the application specialist that there may be problems. If the two application definitions use common objects, only one copy of that object instance will exist in the object base and will be shared by both application definitions. Since two different application definitions are controlling a single object, the object may enter an unexpected (and undesired) state from the perspective of at least one of the application definitions. Architect cannot detect this problem before the new application is parsed; it cannot "look" into the file on disk to see which objects the new application uses. Architect does not "know" which objects the new application uses until the file is parsed into the object base, but then it is too late to recognize a problem; one of the duplicate objects will have already been overwritten.

4.3.2.5 Building Subsystems from Patterns of Control Finally, different classes of subsystems may follow a pattern of control common to its subsystems. The capability to build subsystems with assistance allows the application specialist to create a new subsystem following some predetermined configuration. The system can make simple suggestions

or offer several alternatives to help build the subsystem. Ideally, the knowledge to build these subsystems would be encoded in a knowledge base and used whenever appropriate. However, this type of expert-system is outside the scope of this thesis.

4.3.2.6 User Interface Since the interface will eventually become visual, we used REFINE rules, rather than a menu, to keep the implementation simple and easy to extend. Every option that is applicable at any given time will appear when the application specialist performs a rule search. He can then apply whichever rule corresponds to the action he wishes to take. Flags will be used to prevent the user from being able to apply rules in an inappropriate sequence.

4.3.2.7 Top-Level Object A high-level object will connect all the objects for an application into a single REFINE abstract syntax tree. The user will give the name, corresponding to the name of the application definition, for this object. This will allow the application specialist to manipulate several different application definitions. This also allows for all the objects comprising an application definition to be grouped so they can be saved as a group. This is *not* same as the application object described in Section 4.7.

4.4 Complete Application Definition

After all of the application specialist's input is parsed into the object base, additional processing may be needed to complete the definition. Generics must be instantiated to create specific object instances, incomplete objects must be completed, and objects must be physically loaded into the object base.

4.4.1 Instantiate Generic Objects Each generic specification has three parts: an object instance (i.e., a subsystem or a domain-specific primitive object), a list of generic parameters or placeholders (including an id and a type), and list of internal objects, as shown in Figure 4.3. To instantiate a generic instance, Architect first checks that the instance is valid, i.e., it must reference an existing generic object and it must have the correct number of parameters of the correct type. If this generic instance is valid, the system makes a copy of the component object, then goes through the list of generic parameters,

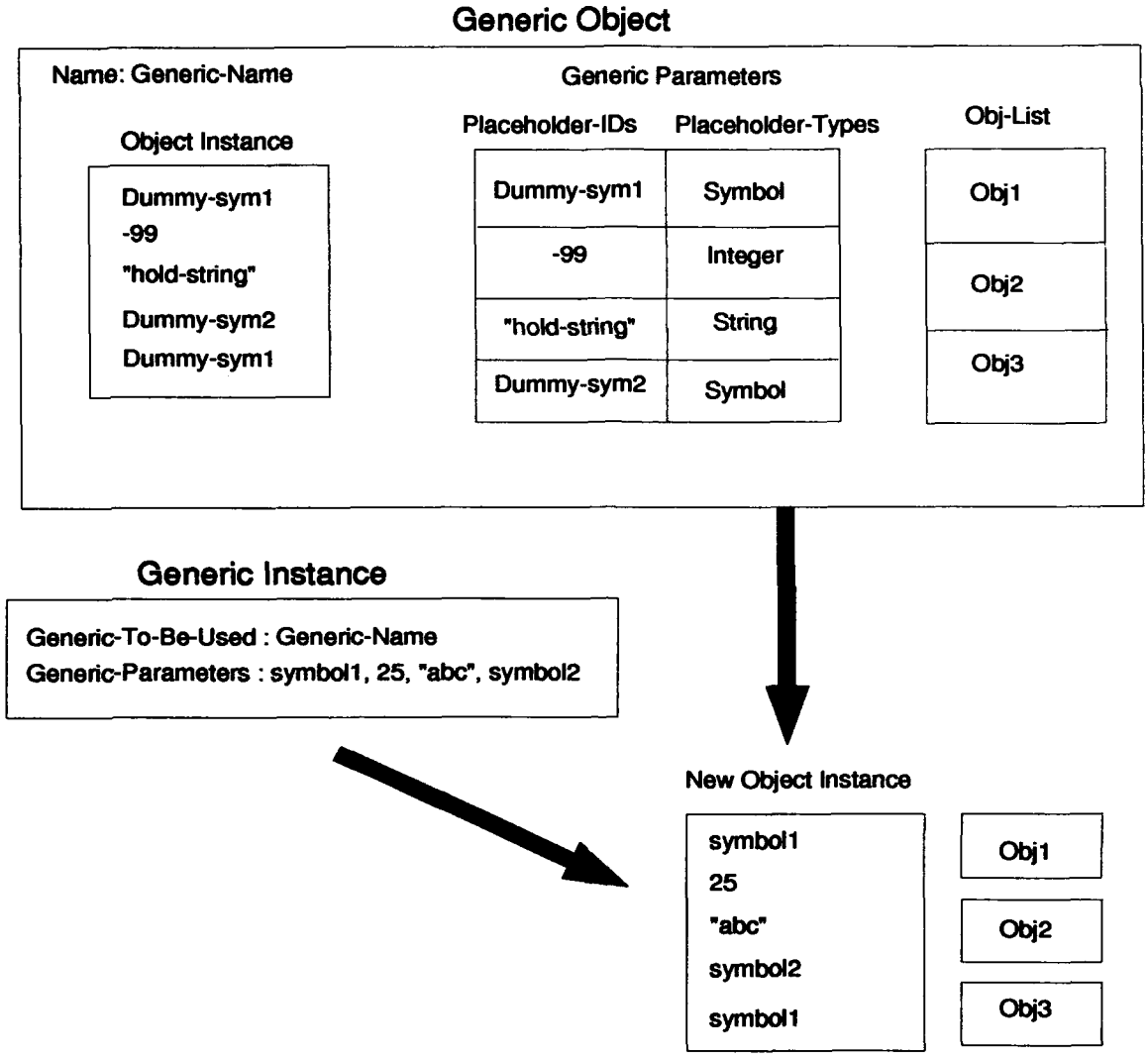

Figure 4.3. Generic Instantiation

finds the applicable placeholder in the new object instance, and replaces each instance of the placeholder value with the new value from the parameter list. If the generic uses internal objects (listed in obj-list), they are parsed into the object base and assigned to the current application definition. The result is a new object in the application definition with the name specified by the application specialist. If the generic instantiation is not valid, Architect will allow the application specialist to edit the object to fix any problems. If the application specialist decides to abandon this generic instantiation, the system will erase the object completely and take no other action. If he does successfully edit the object, it will be instantiated as described above.

4.4.2 Complete Incomplete Definitions The application specialist has the option to incompletely specify an object instance when inputing through a grammar. This allows templates to be saved with only the object type and name specified; specific data for each object instance can be tailored based on the application being specified. After the input is parsed into the object base, all the incomplete object definitions must be completed. Architect checks that the incomplete definition is valid; that is, the incomplete object must have a unique name and it must specify a valid object class. If these conditions hold, Architect creates a new object of the specified type and prompts the application specialist for all required data for the object instance. This new object becomes part of the application and the original incomplete object is erased. As with the generic instances, if the incomplete object is not valid, the application specialist is given the option to edit the object. He can either fix the problem or erase the object.

4.4.3 Load Object Instances The application specialist may require existing object instances stored in the technology base. As part of completing the application definitions, the object instances requested in the application specialist's input file must be copied from the technology base into the object base and assigned to the appropriate application definition object. These instances will be stored in separate files that must be parsed into the REFINE object base. These objects will be stored in a format determined by the appropriate architecture and domain-specific grammars so that they can easily be parsed back into the object base using the REFINE `parse-file` function. Again, if the load object contains an error, the application specialist can edit or erase the object.

4.5 Edit Object Base

The application specialist can modify or delete specific object instances in the object base. If this capability is not available, he would be required to make the changes to his input file (provided at least part of the input was parsed through a file), re-parse the file, and interactively make the same changes as before, except for the object(s) to be modified. Also, Architect must allow for additional object instances to be added or objects to be removed from the object base. Since these changes may effect the integrity of the

application definition, preprocessing and semantic checks must be reaccomplished before simulating execution or generating a specification.

Aside from the objects used to implement the Object Connection Update (OCU) model, the objects are domain-*dependent.* Architect cannot contain any information about specific domain primitive objects except what it can derive from the domain model. As a result, editing an object might not be straightforward from the application specialist's perspective. Architect cannot provide useful prompts beyond the attribute name and the type of data required. If the attribute name is descriptive of the type of data required, the lack of prompts will not be a problem for the application specialist. Also, if any of the attributes are not to be visible to the application specialist, they will still show up when an object of that class is edited. In the case of subsystems, the system can be programmed to ignore these attributes.

4.6 Save To Technology Base

Objects in the object base can be stored in files for later use. The objects are stored in the format determined by the domain-specific and architecture grammars. This can be done rather easily in REFINE; objects can be printed according to a specific grammar that defines their structure. Objects can simply be printed to a file and later parsed into the object base. If an entire application is to be saved, Architect will enumerate through all of the components and write each object to a file. The file name will be the object or application definition name.

4.7 Object Hierarchy

Figure 4.4 illustrates the hierarchy of objects used to implement this system. All these objects define the architecture model's structure, except Primitive-Obj, which is the superclass for all domain primitive objects. The top-level object, World-Obj, ties together all of the types of objects in Architect. The second-level describes different, unrelated, types of object classes within the system. The Import-Obj, Export-Obj, and Statement-Obj classes are used to describe subsystems and the Generic-Obj class is for generic objects. The Spec-Parts-Obj class contains all of the objects that can be parsed

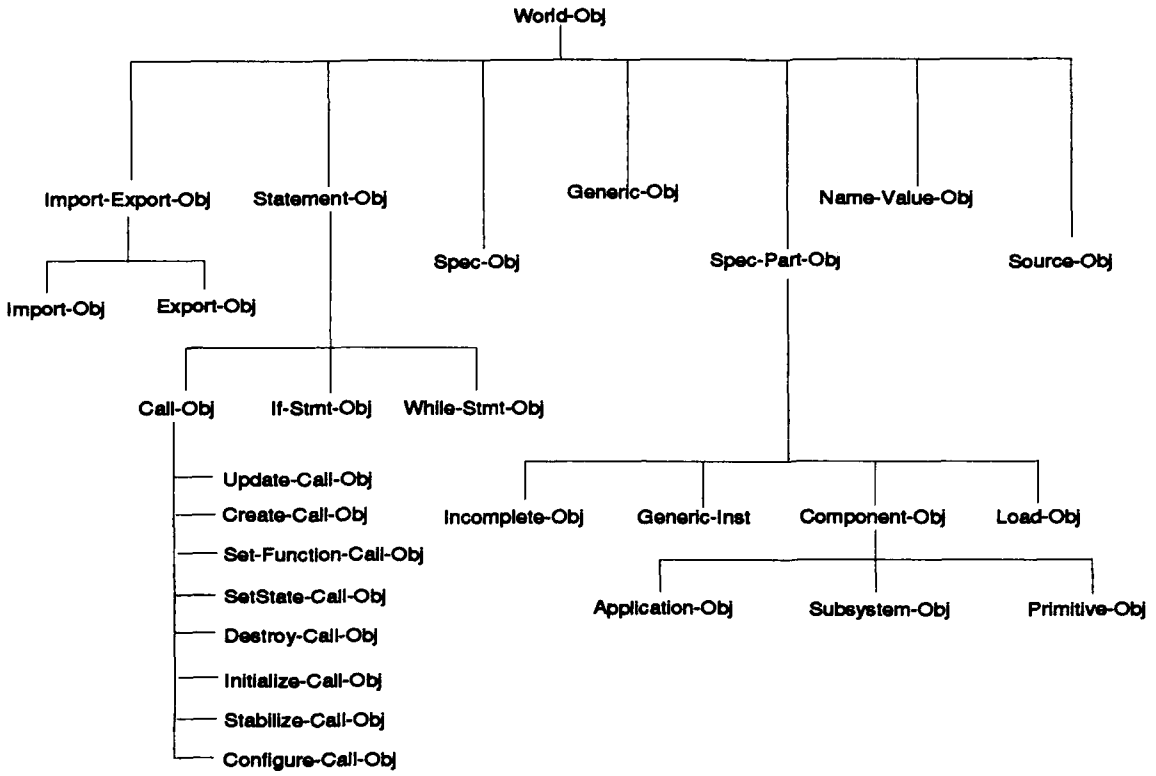

Figure 4.4. Object Hierarchy

through the grammar, including: incomplete objects, generic instance objects, load objects, and component objects. The component objects are either subsystems, application objects, or domain primitive objects; the objects that will ultimately comprise the application definition. Each application definition must contain one and only one application object. This object is similar to a subsystem in structure and behavior, but it describes the top level functions of the application and acts as the starting point for simulating the execution.

4.8 Data Organization

4.8.1 Technology Base

4.8.1.1 Object Definitions The information about each object class is stored in a single file. This allows easier insertion of new object classes into the domain model because it encapsulates all of the knowledge about an object class in one place. Each file is required to contain very specific information in a specific format so the code can be

independent of the domain model. The code simply generates the variable name based on the operation it is trying to perform and the type of object it is manipulating and retrieves the value from the domain model. The software engineer is responsible for creating these files. (For more information on these files see Appendix A.)

An alternative organization would be to combine all of the same type of data in a single file. For example, all the attributes would be in a single file, inputs and outputs would be in another file. If the goals of the system were different, this alternative would be more attractive. It could allow the user (probably a software engineer) to build new primitive objects by selecting the components needed to build the new object. When the technology base extender is fully developed, this design may be more effective.

4.8.1.2 Object Instances, Applications, and Generics The software engineer will store object instances, application definitions, and generics in separate files that will be parsed into the object base when they are required. Like the specific instances, they will use the current domain-specific grammar. The file name will be the same as the name of the object being saved with the appropriate suffix.

4.8.2 File Organization

4.8.2.1 Storing the Technology Base The files that comprise the technology base are stored in a subdirectory such as "tech-base." The software engineer can set up a directory structure to store different domains in different directories. The actual name of the subdirectory is not important; the only time it is used is when loading the files.

4.8.2.2 Storing Other Files The OCU model does not describe any categorization for subsystems. This means that the only lines upon which to organize subsystems are domain-specific. Since Architect itself must be domain independent, no specific scheme can be built in. None of the other data stored in the technology base have a domain-independent classification either except for the type of data they represent (i.e., a generic object, object, or application definition). We can provide some "tools" that allow the software engineer and application specialist to determine categories of subsystems. Copies of

saved instances and generics can be classified and stored in different subdirectories. This requires a common directory for each specific object type; e.g., all generic objects must have a common root. This is set up by the software engineer by specifying a path name. These path names can be any legal path name, either absolute or relative to the current directory. It will be the responsibility of the software engineer to ensure that the corresponding directories actually exist. Architect stores the paths of these root directories and will use these variables when accessing the files. The application specialist can indicate the category and the name of the object to be saved or retrieved by specifying a subdirectory.

4.9 Unique Names

If multiple instances with the same name were allowed in the object base, Architect would not be able to determine which instance to update. Therefore, the object instances' names must be unique. REFINE includes a **unique-names-class** function that ensures all objects within that class are unique. We require all objects to have unique names; a subsystem cannot have the same name as one of the domain primitive objects. Otherwise, whenever the system accesses an object, it must know its specific class. In REFINE, the unique names function does not include subclasses.

4.10 Summary

This chapter outlined the design goals and the specific design decisions made for this research to meet the requirements defined in Chapter III. The overall system design was first discussed and was followed by the particulars of this effort. The steps required to first populate a domain-specific formal object base were described as were the methods of modifying the object base.

V. *Validating Domain*

During initial development, we used a simple, artificial domain to prevent us from being influenced by any biases that a more realistic domain may include. Before development was completed, we moved to a more realistic domain to validate our work. This chapter explains this validating domain and what was learned during its implementation.

5.1 Description of Domain

The domain we chose was digital circuits. It is a well understood, clearly defined field and the expected results of simulating the execution can be easily determined to verify the behavior of Architect. In this domain, simple gates and other low-level components are the primitive objects. Combinations of these components can be built to simulate high-level components. The application definition itself simulates an integrated circuit.

5.1.1 Primitive Objects The domain contains many potential primitive objects. We selected to model those objects which would create a good test domain for our system. Limited domain analysis was performed to develop a domain model because the main goal was to create a domain to exercise the capabilities and to uncover potential problems with Architect, not to build a complete model of the domain.

The primitive objects that comprise the initial digital circuits domain are:

- And gate
- Or gate
- Nand gate
- Nor gate
- Not gate
- JK flip flop
- Counter
- Switch
- LED

All of the gates are modeled as having two inputs, even though other configurations are possible. The switch and LED are used for initial input and output, respectively. The counter was added to exercise the use of coefficients, with the coefficient representing the counter's range.

Just as with the world of digital logic, using only very low level components makes composition of circuits difficult and tedious. Our initial application definitions tended to be long, just to create a relatively simple circuit. Adding additional primitives allows specifications to be written more easily and more clearly. We decided to include the following additional primitives that model real-world digital components:

- Half Adder
- 3 x 8 Decoder
- 4 x 1 Multiplexer

These additional primitive objects allowed for more complicated test cases, such as a 2 x 2 binary array multiplier, Binary-Coded Decimal (BCD) adder, and a Read-Only Memory (ROM). Before adding these new primitive objects, we had built a half adder and 3 x 8 decoder using the original primitive objects. Having these new primitive objects allows for comparison between functionality as well as ability to create more complicated application definitions.

5.1.2 Creating a DIALECT *Domain Model* All of the primitive objects in this system follow a similar template to that of Figure A.1 in Appendix A. The DIALECT domain model for this domain was implemented following this format. Once all of the information was identified, creating the files was straightforward. The final DIALECT domain model is showed in Appendix C.

5.1.3 Update Functions The update functions of each primitive object simulate the behavior of its corresponding hardware component. Since Architect is capable of having multiple update algorithms for each primitive object, at least one primitive needed to have two or more update functions to demonstrate this capability. The LED has a second update function to display the output differently.

5.1.4 Inputs and Outputs Once the primitive objects were selected, the input and output data of each were determined. Both the inputs and outputs were directly determined from the hardware counterpart of the object being modeled. In this case, the category and type of data input and output from each is the same respectively: a signal category and boolean type. The names of the inputs and outputs follow common digital logic symbology as much as possible.

5.1.5 Creating a Domain-Specific Grammar The domain-specific grammar followed from the DIALECT domain model and some additional knowledge about the domain. The general format of each production were the same, with the differences being special attributes, such as boolean-valued attributes, and optional attributes The lessons learned from developing this grammar are documented in Appendix A as recommendations and direction for the software engineer. The grammar developed for this domain is listed in Appendix C.

5.2 System Changes

In the process of transitioning to a new, more realistic domain, we discovered some changes that needed to be made to Architect. Some of the changes were uncovered because of additional experience with the system, not just because of the new domain itself.

5.2.1 Import Sources The area of the most significant changes involved the import areas. The definition of how the sources of the import objects were derived came under close scrutiny. This occurred because the original artificial domain included several possibilities for import sources, but the new domain required all of the data passed between objects to be the same type, the worst case scenario. Since the system could not narrow the possible sources of an import object, all possible interconnections were valid.

Defining these interconnections can, at times, be very tedious. Once this is done, the application specialist will not want to repeat this step unless necessary. However, if he saved his work to a file and later retrieved it, these interconnections would have been lost. The grammar was modified so that the interconnections are also saved.

The idea behind using generic objects is to reuse templates. The interconnections among objects in a generic can be part of that template. Initially, generic objects did not include this information, but now, if the software engineer chooses to include this, it can be added to a generic object. The overall workings of the generic objects have not changed; the procedures for building them has. Changing the grammar to store the imports and exports with objects also allows this information to be stored with generic objects.

5.2.2 Generic Objects Along with the changes mentioned in Section 5.2.1, other changes were made to the generic objects. Since generic subsystems may use objects that do not have to be generic parameters, these objects can now be saved with the generic object and automatically loaded into the application definition when the generic object is used. This does limit the usability of the generic objects that use this capability since they can only be used once per application definition. The software engineer has the option to include specific object instances or not when building generic objects.

5.2.3 Domain-Specific Grammar Transitioning to the validating domain demonstrated some of the issues that must be considered when developing a domain-specific domain model and grammar. The artificial domain initially implemented used few attributes, which were all mandatory, so the grammar did not need keywords. In the digital circuits domain, we included more attributes, many of which are optional. The easiest way to implement a usable grammar is to use keywords. However, we discovered that if the keywords are not chosen carefully they may overlap potential inputs, causing the input to parse incorrectly. As a result of implementing the domain model and grammar for this domain, we developed templates and recommendations for building the DIALECT domain model and grammar. These templates and recommendations are outlined in Appendix A.

We also added the capability to include comments in the applications specialist's input files. The main reason for including this is to aid in testing, but additional information in the file may be helpful to the users of Architect.

5.3 Limitations of the Domain

This domain provided most of the features required to demonstrate all of the capabilities of Architect, but it was lacking in some areas. We modified the domain to create specific situations to exercise specific sections of the system.

5.3.1 Not Representative of All Domains It is impossible to find a single domain that can completely validate the performance of Architect. Domains, by definition, are different, so implementing a single domain will show many aspects of the system, but it cannot be viewed as representing all possible domains. For example, this domain represents closed systems; they do not require external input that is needed by a transaction-oriented domain such as a bank teller system.

5.3.2 Attributes The primitive objects do not have very many natural attributes, so we added several attributes of different data types to show the implications on the grammar and interactive interface. If this were a true domain model, many of these attributes would be considered extraneous.

5.3.3 Varying The Update Function Another problem was with the lack of variations in the update functions. The OCU model allows for an update function to be altered slightly through the use of coefficients or to be completely replaced with another update function. None of the original proposed primitive objects were natural candidates for coefficients so we added a Counter primitive object that used a coefficient to set the maximum count. We also added a secondary LED update function that prints the result differently to demonstrate an alternate update function. If the domain model is required for more than a proof of concept demonstration, it will require modifications to make it more realistic.

5.3.4 Import Source Types Another limitation of this domain is the lack of different types of data to be passed between objects. Every primitive object has the same type of input and output – boolean. The inputs and outputs for each primitive object include a category that will limit the potential sources for an import object of a subsystem. In this domain, however, this additional capability goes unnoticed since it never applies; all

data is modeled as a signal. Having all the same type data being passed around led to us concentrating on ways to improve the usability of Architect based on experiences from this domain, but these "improvements" may be hindrances in other domains.

5.3.5 Composition Constraints A good architecture model must be able to restrict the user from incorrectly composing components that are incompatible. In this domain, no such restrictions exist since any primitive object can be composed with any other primitive object. This domain did not allow us to demonstrate the full capabilities of the architecture model.

5.3.6 Lack of Varying Functionality Some of the benefits of using generic objects and incomplete objects went unnoticed. The concept behind using generic objects is to provide a template for components. This can be useful in building different components using the same template but possessing different functionality due to differences in the parameters. However, In this domain, like objects show very little variation in functionality or level of detail because they have no attributes that affect the execution. If a generic object uses a 3 x 8 decoder as a parameter, all 3 x 8 decoders will have the same behavior so each instantiation of this generic object will function the same. This also impacts the use of incomplete objects in the grammar. If the attributes did affect the behavior of a component, incomplete objects could be used in an application definition template where the final behavior is determined by the attributes entered. With the digital circuits domain, the different attributes often have only trivial effects on the behavior.

5.4 Limitations of the Implementation

5.4.1 Timing and Concurrency The digital circuits domain includes specific constraints, such as timing delays and concurrent execution that our system cannot currently model. For example, given a gate with two inputs, the output of this gate depends on its inputs. However, if the inputs do not arrive at the same time, the output may be different for a short period of time. We currently cannot model this delay, the gate will act as if both inputs arrive simultaneously. REFINE itself does not support parallel processing and Architect currently does not attempt to simulate this behavior.

5.4.2 Restrictions on Primitive Objects In the realm of digital circuits, several of the gates could potentially have more than two inputs. These must be modeled in Architect as separate primitive objects. For example, a two-input And gate could be one primitive object, but a four-input And gate must be a separate primitive object. There cannot be a single And gate object that allows a different number of inputs in different cases even though other primitive objects could allow for similar variations without requiring separate primitive objects. The reason for this problem is that the inputs and outputs for each primitive object class must be incorporated in the domain model. The OCU model uses the object inputs to build the import area for a subsystem. Each of the objects in the import area must have a corresponding export object, otherwise the application definition is incorrect. If this restriction is eliminated, inputs that are actually required may not have a corresponding source. The resulting errors will not be detected until the execution is simulated.

5.4.3 Flat Domain Model Each attribute name has the object class name prepended to ensure that all attributes are unique. However, this does cause a limitation on the structure of the overall domain model. Basically, the domain model must be flat, that is, it cannot take advantage of an object hierarchy. In our digital circuits domain, all of the gates could be subclasses of a gate object class and inherit all of its attributes. Instead, objects that represent subclasses of an object class are modeled separately and all of the common attributes are repeated. Otherwise, attributes of the higher level object class would not be recognized as attributes of the primitive objects in subclasses since they do not begin with the object class name.

If the requirement to prepend the object class is lifted and the code modified, several other changes will be required. With the addition of new object classes, the system must be able to identify which classes can be used to create valid objects. For example, if the domain model includes a gate object class that has And, Or, Nand, and Nor gates as subclasses, an application definition cannot include a "gate" object since the domain does not include this type of component. In other cases, the superclass may represent a valid primitive object class. So if this hierarchical domain is permitted, the code must be

modified and the decision must be made concerning how to prevent invalid object classes from being used to build primitive objects.

5.4.4 No User-Defined Types To edit objects, Architect must be able to recognize all of the possible REFINE data types. The edit function currently recognizes most of the REFINE data types, specifically: integers, reals, booleans, symbols, strings, sets, sequences, any-type and objects. It does not recognize tuples or characters. If the software engineer defines a new type, Architect will not be able to modify any attributes of that type because the function will not be able to determine the type of the attribute. Execution of the `SetAttribute` function only recognizes integers, reals, booleans, symbols, and strings, further limiting the possible types that can be used. The types that Architect makes available to the software engineer and domain engineer represent the basic data types in computer science and should be adequate to describe all the primitive objects of a domain.

5.4.5 Unknown Types If the application specialist adds a `SetAttribute` function call interactively, Architect does not know the data type expected since it can be any valid type, depending on the attribute being set. When the application specialist parses or interactively reads in values for `SetAttribute` function, non-numeric values can either be represented as a string (includes quotes) or as a symbol (no quotes). If the attribute value is a symbol and the system represents non-numeric data as strings, or if the value is a string and the system represents non-numeric data as symbols, a semantic error will occur. This situation can be avoided by representing all non-numeric data as either strings or as symbols and making the code work accordingly. This puts a further restraint on the types of attributes, but character data can easily be represented by all string values or all symbol values.

5.4.6 No "Black-Box" Components A subsystem in the OCU model describes all the data it requires and uses, but it does not indicate which of these data are internal and which are external. Without this capability, application specialists cannot treat a non-primitive component as a "black box." When the application specialist performs the semantic checks, Architect will prompt him to identify all of the sources for the imports,

including the imports within the components created and loaded into the application previously and components built from generic objects. If a source already exists, the application specialist may elect to keep the current source so he is not required to re-specify these interconnections, but he must be sure to change the external connections. In contrast, if the component is modeled as a primitive object, the details concerning its structure are abstracted out. The application specialist is only concerned with how the component interfaces with other components in the application.

In an effort to compare the impact of using the new primitive objects (Half Adder, Decoder, and Multiplexer), we built two binary array multipliers, one using the new Half Adder primitive, the other using two instances of a generic Half Adder. The application definition using the generic Half Adder was longer and included more primitive objects, as would be expected. Specifying objects was relatively easy. The difficulty came when indicating how the objects were connected. During the semantic checks, when the object import sources were identified, the application definition using the generic objects required a total of 30 connections: 16 new, 6 confirmed (internal to the generic instantiations), and 8 changes (external to the generic instantiations). In contrast, the application definition using the new primitive Half Adder required only 16 connections. The second application definition obviously is much less likely to contain errors. The execution of each application definition was identical.

5.4.7 Incomplete Error Checking One of the requirements of Architect is that all object names must be unique. REFINE has several tools that will help maintain a correct object base, but it cannot detect certain actions that may lead to an inconsistent object base. REFINE can prohibit objects of the same class parsed from an input file from sharing the same name. This is set up in the object class definition file in the DIALECT domain model using the **unique-names-class** function. However, this only prevents objects of the *same* class from sharing a common name. This system requires that all primitive objects and all subsystem objects have unique names since it often does not know the specific object class of an object when it tries to find it in the object base. If two different primitive objects use the same name in the object base, Architect will not know which one

to use. If an object is entered interactively, the system checks that no primitive object or subsystem object exists with the new name so this problem will be averted.

This inconsistency in the object base can occur in other situations also. Generic objects and saved subsystems may contain object instances. When these generic objects are instantiated or when these subsystems are loaded into an application, Architect cannot check that the new objects that will be added have names that are not used in the object base already. The system has no knowledge of what objects are going to be loaded until they are loaded into the object base, when it is too late to warn the user. The only indication that a problem may exist is if a semantic error results (e.g., an object is used in two subsystems) or if REFINE displays a message that it is redefining an object.

5.5 Domain Analysis

Domain analysis is not specifically part of Architect at this time, but it is required to define the domain model. Using an artificial domain did not demonstrate some of the potential impacts of domain analysis on Architect, but a real domain did. For example, the process of domain analysis should cover an entire domain, but not all of the components should necessarily be modeled in the system. The goals of the overall system must be considered when deciding which components to model. This domain contains many similarities to the hardware world it models. If a hardware designer only has very low level components such as Nand gates, he has the capability to design very complicated circuits. However, the circuit will be very complex and much more prone to errors. Just as adding higher level components, such as Adders and Multiplexers, aids the hardware designer, adding higher level software components to a domain model aids the application specialist. He can abstract out unimportant design issues by using a high level component and concentrate on just its interconnections.

5.6 Summary

The digital circuits domain served well as our validating domain; it brought out some of the weaknesses or potential weaknesses of the current implementation of Architect.

Perhaps more importantly, it also demonstrated some of the future enhancements necessary for our system to be useful in some domains.

VI. Conclusions and Recommendations

6.1 Introduction

This research has demonstrated the potential applicability of domain-specific languages (DSLs). DSLs can be used as an interface between a sophisticated end-user (an application specialist) and a specification-generation tool. The product of this tool, a formal specification, can then be input into a code generation system to produce an application. Several more specific results concerning DSLs were also discovered in this research as was some direction for future research.

This chapter outlines the overall conclusions and recommendations of this research. It starts with a review of our original goals and then the specific results of this research. The results are classified in three categories: DSL development, domain analysis, and the use of the REFINE environment. We uncovered some shortfalls in our particular implementation of a DSL. These problems should be considered by anyone researching or developing a DSL so that possible solutions can be investigated early in the process to avoid problems in the final product. This thesis is part of the beginning of research into domain-specific software development, so recommendations for future work are given to provide possible direction for future work in this area.

6.2 Original Goals

The original goal of this research was to investigate a means of creating and manipulating a domain-specific formalized object base using a domain-specific language and other required functions. As part of this, we needed first to define the term DSL in the context of the overall research of domain-specific software development, and specifically in context of the work being done to support building an application composition system. This "big picture" approach forced us to investigate how this domain-specific software development system should be structured and where and how domain-specific languages fit in this paradigm. This perspective influenced the research by keeping us from viewing DSLs in isolation.

6.3 Results

6.3.1 Development of a Domain-Specific Language

6.3.1.1 Defined Domain-Specific Language In our context, a DSL is a language that describes (1) instances of domain primitive objects (as defined by the domain modeling language (DML)) and (2) the grouping (composition) of these objects and their operations into higher-level application operations (according to the architecture model) to form an executable application definition, which is the basis for a formal specification. Defining the term "domain-specific language" was one of the first difficulties in this project. Rubén Prieto-Díaz's definition of a domain-specific language (see Chapter II) provided a good starting point, but it did not tell us what was required of, or how to implement, a DSL. Our definition of a domain-specific language is slightly different because of our different perspective. The main differences are the addition of the software architecture structuring rules described in Section 6.3.1.3 and a domain modeling language (DML) explained in Section 6.3.1.2.

6.3.1.2 Incorporated Domain Modeling Language One of the keys to our development of a DSL was the use of a domain modeling language (DML) to describe the domain model. James Neighbors' version of a domain language describes the domain objects and operations, but we describe them using a DML, not a DSL. Our DML acts as a meta-language described in Neil Iscoe's research and formally models the domain's objects and operations. The use of a DML separates domain analysis and modeling from development of domain-specific languages. The DML represents the information gathered from domain analysis as a formal model which then becomes the basis for the development of a DSL. The DSL is still dependent on the resulting domain model and uses the objects and operations defined in the domain model.

6.3.1.3 Incorporated Architecture Model Our DSL is a formal language that serves as a specification language for applications in the domain, but it also includes one aspect not part of DSLs described in the literature or other requirements languages – an architectural model. Common architectures are one of the results of domain analysis,

but rather than incorporating this information in each DSL, we use a general architectural model and domain-specific rules defining valid combinations within the model. The Object Connection Update (OCU) model fills the role of our architectural model, providing the framework for all application definitions, regardless of the domain. The use of a general architecture model gives structure to the resulting specification.

6.3.1.4 Demonstrated That A DSL Does Serve as a Requirements Language

A DSL must be able to specify an application, but does it fill the role of defining requirements? Recall from Chapter II the features Sol Greenspan described as being necessary in a good requirements language (9:3-4):

1. Direct and natural modeling of the world,
2. Support the organization and management of large descriptions,
3. Allow expression of assertions (what is true in the world), entities (objects in the world), and activities (events that cause change).
4. Be uniform in its use of basic principles so that it will be easy to learn and use,
5. Be formal, its features defined precisely,
6. Provide guidance in modeling large-scale problems.

Item 1 is handled by using an object-oriented approach. Objects represent real-world entities and the operations represent the possible behaviors of the objects. Item 2 depends partially on the domain model. If primitive objects in the domain model are built using abstraction principles, the corresponding DSL will also use these features. This points out one of the differences in Greenspan's Requirements Modeling Language (RML) and the DSL developed; our DSL depends on the domain model whereas RML does not. A software engineer using RML builds the state space needed for the application instead of using a defined domain model. Two of the three components listed in item 3 are present in the DSL developed: only assertions are missing. These assertions were not needed to satisfy the immediate goals of this research but will be required before a complete specification can be generated. Entities are the objects, and activities are the associated operations. Even though DSLs are by definition different for each domain, they all share the same basic structure and are built according to the same principles (item 4). A requirements language must be formal (item 5), so we built our DSL based on formal architecture and

domain models so that all the features are indeed defined precisely. Guidance for modeling large-scale problems (item 6) comes from the architecture model incorporated in the DSL. It determines the packaging of components into higher level components. Thus our DSL, when combined with the domain and architecture models, does meet most of Greenspan's features for a good requirements language.

6.3.2 Role of Domain Analysis Our experience with the validating domain demonstrated the flexibility of Architect regarding the domain model. The software places no restrictions on the domain model regarding the level of abstraction of the primitive objects. For example, the initial domain model for our validating domain included only low-level gates such as And, Not, and Or gates, but we were able to build larger components such as decoders and half adders from these primitives and use them in larger circuits. After we added half adder and decoder primitive objects, we were able to replace the low-level primitives with these new objects and maintain the same behavior in the application. The domain engineer and software engineer must build a domain model that contains the appropriate levels of abstraction required to build applications within the domain. Then, it is the responsibility of the application specialist to use the appropriate primitives in each application definition to achieve his objectives.

6.3.3 REFINE *Environment Is Conducive to Building Prototypes* Overall, the Software Refinery Environment provided a powerful, flexible platform for this research. The tools in the environment: the REFINE language, DIALECT, the object base, and INTERVISTA, each contributed to some aspect of the research. Having this integrated tool set allowed us to focus on each specific task, developing a language or writing an application, without having to worry about interfacing with other modules.

Using the Software Refine Environment did have some disadvantages. In the case of the DML, REFINE has too much flexibility to be used without some additional structure (see Section 6.5.2 for more discussion). This environment is better suited for developing prototypes rather than production systems. It lacks the required efficiency; REFINE is built on Lisp which is an interpretive language so the code is not executed directly.

6.4 Problem Areas

6.4.1 DSL Cannot Handle All Grammar Constraints Directly Some constraints are best implemented through the grammar, while others are best done through a separate semantic check. While still using our initial artificial domain problem, we had two grammar constraints, one that disallowed having more than one of a particular object in a subsystem, and another that prohibited consecutive `if` statements. We implemented the second (no consecutive `if` statements), but it required us to change the DIALECT domain model by adding an additional intermediate object. This is because DIALECT itself has additional constraints on the format of the productions in the grammar, namely that all symbols on the right side of a production must map directly to attributes of the object. Even though the productions are written using regular expressions, each part of the regular expression must map to an attribute of the object. A single attribute cannot be described by a complex regular expression. This wasn't a problem until we tried to go through the update function and execute the statements. We now had an additional layer requiring an additional check before we could look at the statement. The same thing would have happened for the other constraint. In the case of no consecutive `if` statements, the changes had to be made to portions of the DIALECT domain model describing the architecture model. However, this cannot be allowed since this portion of Architect must remain domain-independent.

These constraints and other similar restrictions cause problems for two main reasons. The constraints are either context-sensitive or they cannot be handled directly by the DIALECT grammar structure (as described above). In the first case, the grammar would become context-sensitive to handle the constraints. Allowing a context-sensitive grammar creates a very complicated language that is difficult to handle. The solution is to test for these types of constraints during the semantic checks. The semantic checks are easier to write, and they can be domain-specific without affecting the domain model. Having a minimum number of restrictions placed on the grammar will make producing a grammar for a DSL much simpler for the software engineer. The disadvantage to this is that the application specialist does not get a report of the error until he performs the semantic checks.

6.4.2 Restricted Domain Model Currently, Architect assumes that an application is part of a single domain, but very often applications cross several domains. Architect assumes that the domain model being used describes the entire application domain. If this is not the case, two or more already established domains can be merged by loading all the object class definition files into the object base and either creating a single grammar that includes all of the productions or using DIALECT's grammar inheritance. In addition to this, the software engineer must merge the semantic checks and resolve any inconsistencies created. One major problem with this approach is that the only domain-specific constraints that are checked are within a single domain since Architect cannot check for inconsistencies caused by merging domains unless the software engineer writes additional functions. Also, we assume that domains are independent, but very often domains can more realistically be described by a hierarchy. If this is the case, it is up to the software engineer and the domain engineer to build the appropriate domain model because they cannot automatically specialize an already defined domain without building a whole new domain model. The domain model itself is limited by REFINE, which does not support multiple inheritance, so object classes cannot inherit from two different object classes.

6.4.3 Simplistic Update Functions Although the OCU model can support multiple update functions for a given object class, it restricts variety between these different functions (at least as it is currently implemented). The implementation currently assumes that all the update functions use the same inputs and outputs, and the sources for all inputs are required to be defined by the application specialist regardless of whether they are used. This is not an issue for the validating domain since all the primitive objects have only one realistic update function. In other domains, this could become a problem. Alternative interpretations of the OCU model could be implemented to fix this situation if required, but they will have impacts. It will take further investigation of the OCU model to determine the best course of action.

6.4.4 Patterns of Control Not Implemented This feature turned out to be very difficult for several reasons. First, it is difficult to define a pattern of control for any domain. Second, to be useful, the overall procedures to build subsystems from patterns

of control must be domain-independent. This means that the patterns of control must all follow some general structure and the domain-specific knowledge must be represented in a domain-independent form. This in itself can be its own research topic: how to represent knowledge. As the OCU model implementation matures, it may generate possible common, domain-independent templates that can be used as patterns of control. More research is needed in knowledge representation to determine a means of storing domain-dependent patterns of control.

6.5 Recommendations for Future Research

6.5.1 Improve Capability for Reuse This thesis cycle will produce the foundation for future work in this area, but more work is needed to completely demonstrate the viability of this type of system. We have focused on building the general framework, but we may have placed too much of a burden on the application specialist. For Architect to meet its potential, it must allow greater abstraction for easier reuse of previous application definitions. The application definition currently must be written at too low a level. The OCU model is intended to be a tool for the domain engineer to use to "capture the patterns of structure and behavior of the real-world subsystems being modeled" (5), but we require the software engineer to build generic templates and the application specialist to either use these templates or build his own subsystems. The application specialist has little guidance on constructing a domain-specific architecture. Part of the problem is that it takes time and experience to build up the domain model and technology base and the short thesis cycle did not permit such a domain model and technology base to develop.

Architect has the tools to build domain architectures – generic objects, existing objects, and eventually, patterns of control – but there is currently no way to effectively combine them for high level reuse. We need a better method of building models for reuse. This will require modifications to the current OCU model implementation. For example, adding a means to characterize inputs as being either internal or external to a subsystem will allow for better "packaging" of a subsystem and greater abstraction since the application specialist is only concerned with external interfaces when reusing a subsystem.

6.5.2 Conduct More Research On Domain Analysis and Domain Modeling Our development of DSLs depends on domain analysis and domain modeling which are both poorly defined. If the domain model is not built properly, the corresponding DSL will also be incorrect. More work should be done in these areas, specifically; to investigate other domain modeling languages and tools. REFINE is very effective in describing the domain model, but it can be difficult to work with since it is so flexible and powerful and does not include any constraints or guidance on building a domain model. Potentially, a tool similar to KIDS could be built over REFINE to help model a domain. Another approach is to develop a meta-model as described by Iscoe to define the overall structure of a domain model. This meta-model is then instantiated into a domain model using domain knowledge. Since all of our domain models have the same operational goal, they should follow a similar pattern, so this could be a viable option.

6.5.3 Investigate Expanding Current Architecture Model and Other Possible Models More work needs to be done to completely define the OCU model. As it evolves, the changes must be incorporated into our software. This research has demonstrated some of the strengths as well as weaknesses of the current definition of the OCU model that should be considered for future OCU development. Other architecture models should be investigated to determine if they could replace our implementation of the OCU model or if they could influence the OCU model development.

6.5.4 Expand Visual System The visual system is undergoing development as this research is being completed. This area is very important to Architect and should be expanded. The current implementation of Architect using only a grammar and simple interactive interface is much more difficult to conceptualize and manipulate than necessary. A visual system can reinforce the concept of building an application definition specification from existing models by showing the user the building blocks.

6.5.5 Build Technology Base Extender The research done in this area will benefit the development of the technology base extender described in Chapter III but not yet implemented. Building and extending a domain model (and technology base) should use

the same techniques so the research can be used in both areas. Early in our research, we questioned the usability of a DSL to the software engineer to extend the technology base. To some extent, the DSL can be used by the software engineer in this role; he uses the DSL to build components and generic objects. However, the domain model is built using the domain modeling language, not the domain-specific language. The best approach to extending the technology base is to look at the implementation of the DML, not the DSL, to see how it can be used to not only build an initial domain model, but also to extend the model.

6.5.6 Incorporate More Use of Domain Knowledge The role of domain knowledge in this type of system cannot be overstated. Our current system does not use this information nearly to its full potential. Domain knowledge is incorporated into the primitive objects and operations in the domain model and in the domain-specific semantic checks, but it can be used in a much more powerful role. According to Kelly and Nonnenmann, "domain knowledge is used to correct routine omissions and errors in scenarios, constrain the space of possible scenario generalizations, shape the structure of models used in model based reasoning, and plan queries posed to its human oracles" (13:43). To be used, this domain knowledge must be codified in some form useful to the system, i.e., we must find some representation.

This knowledge representation is required to implement the patterns of control discussed in Section 6.4.4 and domain-specific semantic checks. Currently, Architect can easily support the software engineer in developing the simple semantic checks, but it provides no guidance or framework for more complicated checks. A structured rule base, whose rules are populated by the software engineer, and an inference engine to apply the correct rules appropriately is needed. Our use of domain knowledge has not reached this level of sophistication, but this should be a long-term goal.

6.5.7 Consider Draco Approach for Further Development James Neighbors' domain description for his Draco approach includes a parser, prettyprinter, transformations, components, and procedures. We currently have implemented a parser and prettyprinter (built using DIALECT). The transformations, components, and procedures are needed as

part of the next stages of development, design and implementation. Much of Neighbors' ideas and techniques do apply to this research, but his research goes beyond what we have implemented so far. As Architect evolves, Neighbors' work can help define steps that need to be taken and how best to implement these steps.

6.5.8 Add Invariants Greenspan's definition of a good requirements language includes assertions or invariants (what must always be true in the system), but Architect currently does not model these conditions directly. Two different cases can arise: domain-specific invariants or application-specific invariants. The domain-specific invariants must be modeled as part of the domain model and the application-specific invariants must be included in the DSL. To do this, the first step is to find a consistent representation for all invariants. The domain-specific invariants are then stored with the domain model and the application-specific invariants must be included as a new sentence in the DSL grammar. Since these invariants may affect the behavior of the prototype, they should be checked during execution. These invariants are needed to generate a complete specification.

6.5.9 Add Exception Handling Another aspect missing in Architect is exception handling. If the execution step comes across an error, it is not handled except possibly by the update algorithm. However, not all exceptions will be raised during a procedure call that can handle it; the exception could be a side effect.

6.5.10 Add External Interfaces The OCU model supports specification of the hardware interfaces, but this is not currently implemented. In the future, this feature must be further defined and incorporated into the implementation of the OCU model (or whatever architecture model is used). This information will be necessary to generate a specification.

6.5.11 Build a Specification The natural next step in the software development process is to build a specification. The exact implementation of this step depends on the expected use of the specification. If it is to be an input to another automated tool, it must have a precise structure; otherwise, the exact structure is not as important. Several of the components needed to build a specification have already been discussed: external interfaces, invariants, and exception handling. Once these have been added, the next step

is to determine the format of the specification. If anything for the specification is still missing, it must be added at this time. Transformations or other procedures to extract information from the object base are then needed to create the specification.

6.6 Conclusion

The overall goal of this research was not simply to build a domain-specific language, but also to investigate the implications of building a system using a DSL. We first decided what capabilities a DSL must possess, determined how to implement these features in Architect, and then built a prototype. The prototype system brought out some of the weaknesses of our implementation, but more importantly, it demonstrated the effectiveness of using a DSL in this context. A DSL is a tool that lets sophisticated users specify systems, a key feature in program generation or program synthesis systems. We also discovered that a DSL can be combined with a domain modeling language and an architecture model to increase the power of a DSL while isolating specific features to allow for easier modification. It is clear that software development of the future must be improved; the lessons learned from this research can provide direction for future research in this important area.

Appendix A. *Software Engineer's Responsibilities*

The software engineer has several responsibilities in keeping the system functioning properly. He must build and maintain the domain model, grammar, and semantic checks for each domain; architecture model domain model, grammar, and semantic checks; directory structure; and load files. This appendix outlines these responsibilities. It assumes a working knowledge of the REFINE environment.

A.1 Building Domain Model

Several pieces of information must be present in the object definition file for each domain primitive object for the system to run correctly. The following explains this required data. Figure A.1 shows a sample template of a domain primitive object definition. In all cases, "OBJECT-CLASS" is replaced with the appropriate object class name.

A.1.1 Inputs and Outputs Each object class must contain definitions of the inputs and outputs for each object instance of that class. These definitions must be specifically named – the object class name followed by "–INPUTS" or "–OUTPUTS." Figure A.1 shows an sample object class with one input and one output. If the object class has more inputs or outputs, the **set-attrs** command is repeated for each additional input or output. If the object has no inputs or outputs, the corresponding variable is set to the empty set ({}).

Each input and output has three parts: name, category, and type-data. The name is used to determine a specific input or output if more than one exists. The category refers to the general type of data it consumes or produces. This is used to limit the possible sources of data that can be used. For example, the category could be temperature or time. The final piece, data-type, refers to the specific data type, i.e., real, integer, symbol, string, etc.

A.1.2 Coefficients and Update-Function The following two entries in Figure A.1: OBJECT-CLASS-COEFFICIENTS and OBJECT-CLASS-UPDATE-FUNCTION are attributes required for each object definition. The first sets up the coefficients used for each

```
!! in-package("RU")
!! in-grammar('user)

 var OBJECT-CLASS       : object-class subtype-of Primitive-Obj

 var OBJECT-CLASS-INPUT-DATA   : set(import-obj) =
     {set-attrs (make-object('import-obj),
                 'import-name, 'input-name,
                 'import-category, 'signal,
                 'import-type-data, 'boolean)}

 var OBJECT-CLASS-OUTPUT-DATA  : set(export-obj) =
     {set-attrs (make-object('export-obj),
                 'export-name, 'output-name,
                 'export-category, 'signal,
                 'export-type-data, 'boolean)}

 var OBJECT-CLASS-COEFFICIENTS : map(OBJECT-CLASS, set(name-value-obj))
  computed-using  OBJECT-CLASS-COEFFICIENTS(x) =
  {set-attrs (make-object('name-value-obj), 'name-value-name, 'max-count,
                                            'name-value-value, 3)}

 var OBJECT-CLASS-UPDATE-FUNCTION : map(OBJECT-CLASS, symbol)
   computed-using  OBJECT-CLASS-UPDATE-FUNCTION(x) = 'OBJECT-CLASS-UPDATE

% Other Attributes:
 var OBJECT-CLASS-ATTR1 : map(OBJECT-CLASS, integer)
   computed-using  OBJECT-CLASS-ATTR1(x) = 0

 var OBJECT-CLASS-ATTR2 :  map(OBJECT-CLASS, set(symbol))
   computed-using  OBJECT-CLASS-ATTR2(x) = {}

 var OBJECT-CLASS-OPEN? :  map(OBJECT-CLASS, boolean)
   computed-using  OBJECT-CLASS-OPEN?(x) = nil

 form MAKE-OBJECT-CLASS-NAMES-UNIQUE
  unique-names-class('OBJECT-CLASS, true)

function OBJECT-CLASS-UPDATE (subsystem : subsystem-obj,
                              some-obj : OBJECT-CLASS) =
 sequence of calls

function OBJECT-CLASS-UPDATE1 (subsystem : subsystem-obj,
                               some-obj : OBJECT-CLASS) =
 sequence of calls
```

Figure A.1. Sample Object Definition

object instance, the second tells which update function is to be used for a particular instance. Both these attributes are used for the Object Connection Update (OCU) model. (See (1) for more details).

A.1.3 Attributes Next are all of the attributes that describe the object class. As with the other information used to describe an object class, each name must be prefixed by the object class name. An attribute called DELAY will become OBJECT-CLASS-DELAY. All objects have a NAME attribute which is a REFINE defined attribute; it is *not* specified in the domain model. The **computed-using** clause gives each attribute an initial value. This is useful in two cases. The first case is when the attribute value rarely changes, the attribute can be optional in the grammar so the user does not have to supply a value unless it is different from the default. The second case is with attributes that are a set or a sequence. Having these two attribute types defaulted to the empty set or empty sequence frees the code from checking if the set or sequence exists before trying to manipulate it.

A.1.4 Make Names Unique The form MAKE-OBJECT-TYPE-NAMES-UNIQUE makes all of the object instances of the given class unique. REFINE will prohibit any objects of this class to be created if an object with the same name already exists in the object base.

A.1.5 Update Functions Finally, the update functions define the behavior of objects of the specified class. An object may have several different update functions. Normally, they will be similar, differing in some area such as level of precision. Which function a particular object instance is to use is stored in the OBJECT-TYPE-UPDATE-FUNCTION attribute and is set by the application specialist.

A.2 Grammar

The software engineer must develop a domain-specific grammar for each domain that is to be used. This is not difficult, provided the domain was built using the template described in Section A.1. However, he must consider several issues as illustrated in Figure A.2

```
!! in-package("RU")
!! in-grammar('syntax)

grammar Grammar-Name

 no-patterns

 inherits-from OCU

 start-classes Spec-Obj, Subsystem-obj, Application-Obj,
   Incomplete-Obj, Generic-Obj, Object-Class, ...

 file-classes  Spec-Obj, Subsystem-Obj, Application-Obj,
   Incomplete-Obj, Generic-Obj, Object-Class, ...

 productions

  Object-Class  ::= ["Object-Class" name
                     {["attr1" Object-Class-Attr1]
                     [(["is open" !! Object-Class-Open?] |
                       ["not open" ~!! Object-Class-Open?])] }
                      "attr2:"  Object-Class-Attr2  ]  builds Object-Class,
  other domain objects ...

 symbol-start-chars
    "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ./*"

 symbol-continue-chars
    "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ-0123456789./*?"

 comments "%" matching "
"
 precedence

    for expression brackets "(" matching ")"
      (same-level "and", "or" associativity left),
      (same-level "<", "<=", "=", ">=", ">", "/=" associativity none),
      (same-level "+","-"  associativity left),
      (same-level "*", "/", "mod"  associativity left),
      (same-level "not"  associativity none),
      (same-level "abs"  associativity none),
      (same-level "**" associativity right)

  end
```

Figure A.2. Sample Grammar

and described below. The DIALECT User's Guide (23) has more details on how to build a grammar.

A.2.1 Keywords Using keywords for each attribute creates a grammar that is easier to use and much more likely to parse correctly (less chance of shift/reduce or reduce/reduce conflicts). However, the keywords must be chosen carefully so that they do not overlap possible data in the input being parsed. For example, if an object has an attribute called OBJECT-CLASS-ATTR1, it may be used in a **SetState** call. The **SetState** function does not require the entire attribute name, only the name *without* the object class name is needed (e.g., ATTR1). If this short version of the attribute name is used as a keyword, no **SetState** calls with that attribute will parse correctly. One easy way around this is to add a ":" after each keyword so for OBJECT-TYPE-ATTR1, the keyword could be "ATTR1:". Finally, DIALECT requires all keywords to be entered in lower case.

A.2.2 Required and Optional Attributes Not all of the attributes may be required to describe an object. If the attribute is optional, the part of the production that describes the attribute should be between braces ({ }). For example, in Figure A.2, OBJECT-CLASS-ATTR1 and OBJECT-CLASS-OPEN? are both optional, so they are enclosed in braces. In the case of several optional attributes, each keyword-attribute pair should be enclosed in brackets ([ ]) to indicate a sequence of elements. In Figure A.2, OBJECT-CLASS-ATTR2 and **name** are both required. The name attribute must be specified as a mandatory attribute for each domain primitive object, otherwise objects cannot be referenced after they have been built.

A.2.3 Boolean Attributes Another special case is shown in this example, boolean attributes. In this case, the keyword used indicates whether the attribute is set to true or false. If using the grammar shown in Figure A.2, the phrase "is open" will set the attribute OBJECT-CLASS-OPEN? to true while "not open" will set it to false. Boolean attributes will follow the form: (["true keyword" !! attr-name] | ["false keyword" ~!! attr-name]).

A.2.4 Header Information Each domain-specific grammar must have a name which is specified in the **grammar Grammar-Name** phrase. The name can be any valid REFINE

symbol. The **no-patterns** key word indicates that the grammar is to be built without patterns (the grammar may not compile if this is omitted). The **inherits-from OCU** is necessary to include all of the productions defining the OCU model. **Start-classes** and **file-classes** tell which productions can be parsed individually and from a file, respectively. Each of the objects listed (except Object-Class) must appear on these lines, along with all of the domain primitive object classes.

A.2.5 Other Required Elements The rest of the template in Figure A.2 must be used as shown. **Symbol-start-chars** and **symbol-continue-chars** define which characters can begin and continue, respectively, symbol characters. If the software engineer defines an attribute name that uses characters not in these lists, these characters should be added where appropriate. The **comments "%" matching "(carriage return)"** command allows comments to be embedded in the application definition. In this example, comments begin with a % and are terminated by a carriage return. Comments must not necessarily begin with a % nor end with a carriage return; they can begin and end with another character or sequence of characters. If comments are used, the comment delimiters can not be a part of the grammar (e.g., parentheses can not be used for comments since they are used in the grammar). The **precedence** keyword and the following precedence table is needed as shown.

A.2.6 Changing the Grammar If either the domain-specific or the architecture grammars change, all of the saved objects, generic objects, and application definitions may become obsolete. If the change is minor, the files containing the objects can be edited so that they will consist of legal sentences in the new grammar. The alternative is to recreate the objects and save them to the technology base, overwriting the existing files.

A.3 Domain Semantic Checks

Often, the domain model cannot adequately scope a data type to a specific, valid range. If an attribute is an integer between 1 and 10, the attribute must be defined as an integer. If the application specialist defines this attribute to be 20, the object will parse correctly. The solution to this problem is to create simple REFINE rules to check for simple

semantic errors such as the one just described. These rules will be called from the check semantic routine.

A.4 Generic Objects

Building a generic object can be confusing. The best way to attack this process is to carefully define exactly how this generic object is to look. Subtle differences in how a generic object is built can impact heavily how it is used, so the software engineer must know exactly what the application specialists need.

A.4.1 Building Generic Objects

The basic steps to build a generic object are:

1. Build a subsystem or domain primitive object that represents an instance of the generic object being built. If it is a subsystem and the import sources are to be included (i.e., interconnections defined), include all objects that are needed to make it execute.
2. Parse it into the object base as if it is part of an application definition.
3. Build the import and export areas and assign import sources, if desired. This can be done by running the semantic checks.
4. Although not specifically required, the object instance can be built as part of a valid application definition and executed to test that it functions as intended.
5. Apply **`Build-Generic`** rule to indicate which parts of the object are to be considered parameters.
6. Apply the **`Save-Generic`** to save the generic object to the technology base.

When building the instance to be made generic, any of the attribute values can be parameters that are replaced by a specific value designated by the application specialist. It is a good idea to use a descriptive name when possible because the parameter name is used as a prompt to the application specialist if he is entering the generic instance interactively. For example, he shouldn't use a parameter name such as **`and-gate1`**; a more preferable name would be **`And-To-Carry-Output`**. The name should not be one that could be used somewhere in an application definition. If the application specialist uses a name that is a parameter name in the generic object, the system could instantiate the generic instance incorrectly. If the parameter is a number, it is best to use a number that would not be

used normally, such as -99 for a loop variable. It does not make sense to use a boolean value as an parameter.

Each parameter must be uniquely named, but it may reference an attribute that occurs multiple times in the object. An object in a subsystem will appear in the controls list and in the update procedure, which is the case in the simple example shown in Figure A.3. When the generic instance is instantiated, each occurrence of the parameter will be replaced with the parameter specified by the application specialist.

```
generic-obj GENERIC-NAME GENERIC-OBJECT-NAME
  ids: OBJ1, OBJ2, OBJ3, -99
  types: AN-OBJECT, AN-OBJECT, AN-OBJECT, INTEGER
  objects:

subsystem GENERIC-OBJECT-NAME is
  controls: OBJ1, OBJ2, OBJ3
  update procedure:
  while some-value > -99 do
    update OBJ1
    update OBJ2
    update OBJ3
  end while
  update OBJ1
  update OBJ2
```

Figure A.3. Sample Generic Object

Some generic subsystems may require objects that do not need to be parameters. These objects can be defined with, and stored as part of, the generic object. When the application specialist instantiates this generic object, he will automatically get all of these objects in his application definition.

A.4.2 Considerations Several factors must be considered when building generic objects. First, the software engineer must know how a specific generic object is to be used in application definitions so he can build a useful generic object. Frequently the decision as to what should be parameters is not clear-cut and this decision will affect the usefulness of a generic object. If the generic object includes additional objects, it will be easier to use,

the application specialist has less to specify, but it can only be used once in an application definition. Another decision the software engineer faces is where to make the boundary for the generic object. Should it include all related objects, or should they be left to other objects in the application definition?

The system currently performs little checking on the software engineer's input when building a generic object. It does check that the specified parameter does occur in the object and the objects to be included with the generic object must exist in the object base.

A.5 Directories

The directory structure is determined entirely by the users of the system; nothing is hard-coded. If several domain models reside on the same disk, the paths can be relative to the current working domain directory. For example, if object instances are stored as a subdirectory to the technology base, which is a subdirectory to the working domain directory, the path to access these objects should be "./tech-base/objs" (using the appropriate directory names). If the users of the system have, for example, two different categories of objects, they can set up two subdirectories below the root path for objects.

The software engineer will define the root directory for each of the three different types of objects that can be saved: application definitions, object instances, and generic objects. This supplies a central root directory for each different type of data. In Figure A.4, assuming the working directory is "Domain1", the application definitions are stored in "./tech-base/applics," the object instances are stored in "./tech-base/objs," and generic objects are stored in "./tech-base/generics." If "Domain1" is not the working directory, the paths should be adjusted accordingly. These paths are stored with the global variables. If the software engineer wants to categorize the data further, he can create subdirectories of these root directories. For example, he may want to put all object instances in a directory called "objs." If he wants to categorize the objects by type, he can create subdirectories such as "ADDER," "DECODER," etc. The software engineer may also create directories for input files or leave that responsibility to the application specialist.

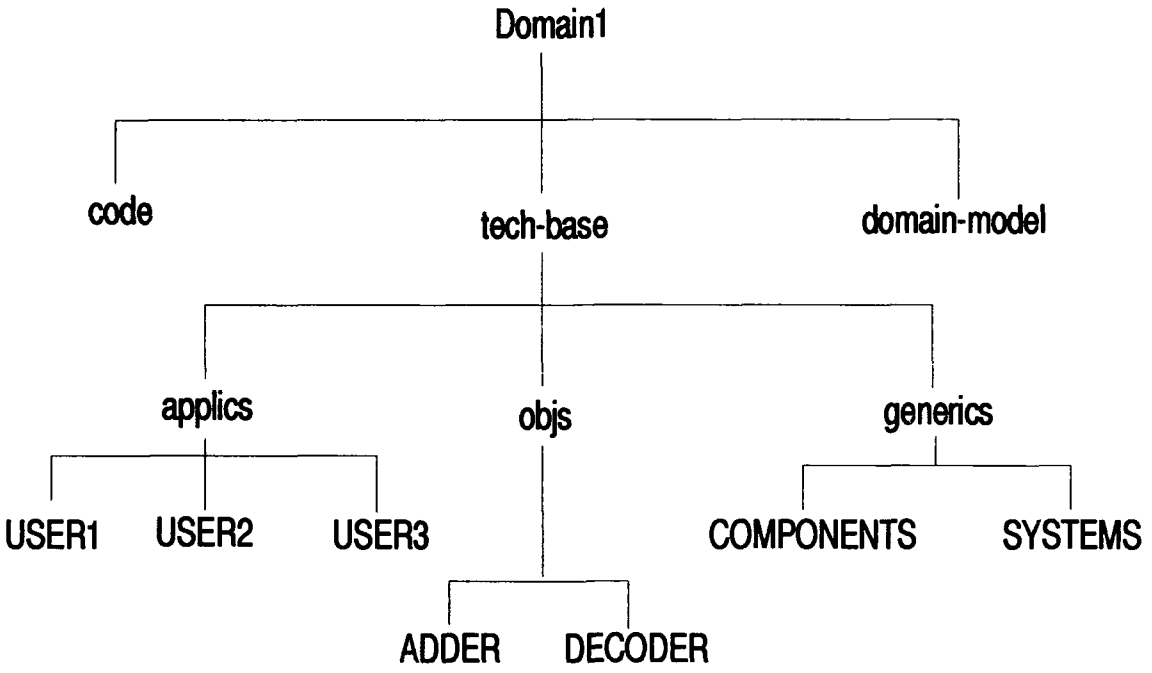

Figure A.4. Example Directory Structure

There is one restriction on the subdirectories to the root directory given by the software engineer: the names must be upper case. The application specialist's input may request objects to be loaded or generics instances to be instantiated. In both cases, he must give the path name if the object is not in the root directory of that type. This path is stored as a symbol, which when converted to a string, becomes upper case. Since Unix is case-sensitive, the directory paths must match the actual directory structure exactly. All path names entered from the prompt will be converted to upper case so they will match the directory structure.

A.6 Load Files

Along with the entire domain model, all of the program files must be loaded into the object base before the system can be used. This can be made much easier by using load files. These files are simply lisp functions that load all of the required files into the object base. This reduces the loading process to two commands: `(load "load-file-name")` followed by the call to the lisp function `(function-name)` to load the system files (parenthesis

included). This load file must begin with a call to load DIALECT (i.e., `(load-system "dialect" "1-0")`) since DIALECT is used by the grammars. The load order is important; as with programming languages, files that include function calls to functions in other files must be loaded before the calling functions. In this system, all of the files that define a DIALECT domain model must be loaded before the corresponding grammar can be loaded. REFINE must be told to change to the new domain-specific grammar. This is done using the REFINE command `(in-grammar 'grammar-name)` after the grammar has been loaded. For the REFINE rule search command to run properly, REFINE must have a defined current node. This can be done in the load file by adding a line `(mcn 'object-name)` where object-name is any object in the object base. Examples of objects that can be used include all of the global variables in the system and REFINE global variables. The application specialist will change the current node once he has parsed an application definition.

Two separate load files should be used, one for the application specialist (see Figure A.5), the other for the software engineer (see Figure A.6). The software engineer requires the additional functions needed to build generic objects so these program files should be included in his load files. The application specialist does not need this capability and should not see the rules for building and saving generic objects in his rule searches.

```
(defun load-files()

  (load-system "dialect" "1-0")

  (load "./OCU-dm/dm-ocu")
  (load "./OCU-dm/gram-ocu")

  (load "./DSL/globals")
  (load "./DSL/lisp-utilities.lisp")
  (load "./DSL/obj-utilities")
  (load "./DSL/read-utilities")
  (load "./DSL/erase")
  (load "./DSL/menu")
  (load "./DSL/display-files")
  (load "./DSL/modify-obj")
  (load "./DSL/save")
  (load "./DSL/generic")
  (load "./DSL/complete")

  (load "./OCU/set-debug")
  (load "./OCU/imports-exports")
  (load "./OCU/eval-expr")
  (load "./OCU/execute")
  (load "./OCU/semantic-checks")

  (load "./domain-model/and-gate")
  (load "./domain-model/or-gate")
  (load "./domain-model/nand-gate")
  (load "./domain-model/nor-gate")
  (load "./domain-model/not-gate")
  (load "./domain-model/switch")
  (load "./domain-model/jk-flip-flop")
  (load "./domain-model/led")
  (load "./domain-model/counter")
  (load "./domain-model/decoder")
  (load "./domain-model/half-adder")
  (load "./domain-model/mux")
  (load "./domain-model/gram-logic")

  (in-grammar 'circuits)
)
```

Figure A.5. Sample Load File for Application Specialist

```
(defun load-files()

  (load-system "dialect" "1-0")

  (load "./OCU-dm/dm-ocu")
  (load "./OCU-dm/gram-ocu")

  (load "./DSL/globals")
  (load "./DSL/lisp-utilities.lisp")
  (load "./DSL/obj-utilities")
  (load "./DSL/read-utilities")
  (load "./DSL/erase")
  (load "./DSL/menu")
  (load "./DSL/display-files")
  (load "./DSL/modify-obj")
  (load "./DSL/save")
  (load "./DSL/generic")
  (load "./DSL/build-generic")
  (load "./DSL/complete")

  (load "./OCU/set-debug")
  (load "./OCU/imports-exports")
  (load "./OCU/eval-expr")
  (load "./OCU/execute")
  (load "./OCU/semantic-checks")

  (load "./domain-model/and-gate")
  (load "./domain-model/or-gate")
  (load "./domain-model/nand-gate")
  (load "./domain-model/nor-gate")
  (load "./domain-model/not-gate")
  (load "./domain-model/switch")
  (load "./domain-model/jk-flip-flop")
  (load "./domain-model/led")
  (load "./domain-model/counter")
  (load "./domain-model/decoder")
  (load "./domain-model/half-adder")
  (load "./domain-model/mux")
  (load "./domain-model/gram-logic")

  (in-grammar 'circuits)
 )
```

Figure A.6. Sample Load File for Software Engineer

Appendix B. *Application Specialist's Responsibilities*

This appendix is intended to provide guidance to the application specialists in creating application definitions and using the system in general. It assumes a working knowledge of the REFINE environment. This appendix describes how the system works as of the end of this particular research, this is expected to change as the visual system is implemented.

B.1 Starting

The software engineer should have file(s) to load the object base already written for each domain. To run the system, start REFINE, then load the load file (`load "load-file-name"`), and finally call the function that loads the files: (`function-name`). The system should then be ready to accept input. (For more information see the REFINE User's Guide (24).)

B.2 Creating New Application Definitions

B.2.1 Building Input Files The input files can be produced in any text editor that can generate ASCII files. The input must contain valid sentences defined in the architecture and domain-specific grammars. The application specialist should be aware of the format for each of the valid sentences; if not, he should have the software engineer explain them. Here is an overview of possible input:

- Build domain-primitive object instances
- Build subsystems
- Instantiate generic objects
- Load existing objects
- Specify incomplete objects

Once this file has been created, it is ready to be parsed into the object base.

B.2.2 Using Generic Objects The system has no built-in knowledge about generic objects so the application specialist must know ahead of time how each generic object can be used. If a generic instance is built or edited interactively, the system prompts for all

the required input, but it will not explain how each piece of data is used in building the new generic instance.

Generic objects, like saved instances, can be stored in different subdirectories to allow for classification. When specifying a generic object in the grammar, the application specialist must give the path of the generic object. All generic objects share a common root directory that will be defined by the software engineer, but generic objects can be stored in any number of subdirectories. If the generic object is in this root directory, only the generic object name is required. For example, if the application specialist wants to use a generic object named **GENERIC-ADDER1** which is stored in the generic object root directory, he only uses the name. If, however, this generic object is stored in a subdirectory, such as **ADDERS**, then the application specialist refers to the generic object as **adders/generic-adder1**. The system converts all paths and file names to upper case so the case in which the names are entered is unimportant.

B.2.3 Using Existing Objects Objects and application definitions can be saved into the technology base for future use. The application specialist has the capability of retrieving these objects. However, since all names must be unique, an object can only be used once in the object base. If an object is needed more than once in an application definition, the application specialist should talk to the software engineer about converting it to a generic object.

B.2.4 Naming Objects All objects must have unique names. If two objects of the same class are given the same name, the system will only recognize one object. If the object is entered interactively, the system will check that no object of that name exists, so the application specialist is prevented from causing problems of this type. The other situations where unique names becomes an issue are described in Sections B.2.3 and B.2.2. The final complication is with generic objects that use internal object instances. Only one instance of a generic object that uses internal objects can be used at a time since the system loads those specific instances into the object base. Again, the application specialist will need to get the details from the software engineer about the generic objects.

```
.> (rs)
- Rules for: ##r USER var FATAL-ERROR: boolean = false -
1) ERASE-APPLICATION-DEFINITION
2) EDIT-AN-OBJECT
3) ADD-AN-OBJECT
4) DELETE-AN-OBJECT
5) SAVE-OBJECT
6) SAVE-APPLICATION
7) LOAD-OBJECT
8) LOAD-APPLICATION
9) PARSE-INPUT
10) ADD-GENERIC-INSTANCE
11) BUILD-GENERIC
12) SAVE-GENERIC
13) COMPLETE-APPLICATION-DEFINITION
15) CHECK-SEMANTICS
.>
```

Figure B.1. Sample Rule Search

B.3.1 Applying Rules Until the graphical interface is developed, interaction with the system is done through REFINE rules. REFINE lists potentially applicable rules when given the **(rs)** command (see Figure B.1). Each rule will be listed next to a unique number. A specific rule can be applied (run) by typing **(ar #)** where # is the number corresponding to the rule or by typing **(ar rule-name)** where rule-name is the name listed (the name must match exactly except case is not important). If the rule search command returns an error, the software engineer probably did not define the current node in the load file. This can be fixed by setting the current node to any object in the system by typing **(mcn 'object-name)** where object-name is the name of a valid object in the object base. Normally, the application definition, or an object that is part of an application definition, is the current object. However, if the object base does not contain an application definition, i.e., no input has been parsed or entered, another object must be set to be the current node. Possible objects that should exist in the object base include: fatal-error, debug-on, re::current-grammar (this is a REFINE object so it should always be available, unless REFINE changes).

B.3.2 Default Values When the system prompts the application specialist for input, it might provide a default value. If it exists, the default value will appear in parenthesis with the prompt. To keep this value, the application specialist simply hits the return key. If he wants to change this value, he enters the desired value.

B.3.3 Functions Available When the application specialist performs a rule search, he will be given the following options (not necessarily in this order) to populate and manipulate an application definition in the object base:

- PARSE-INPUT
- EDIT-AN-OBJECT
- ADD-AN-OBJECT
- DELETE-AN-OBJECT
- LOAD-OBJECT
- ADD-GENERIC-INSTANCE
- COMPLETE-APPLICATION-DEFINITION
- SAVE-OBJECT
- SAVE-APPLICATION
- LOAD-APPLICATION
- ERASE-APPLICATION-DEFINITION

To execute any of these functions, the application specialist applies the corresponding rule as described in Section B.3.1. The following sections describe each of these functions.

B.3.3.1 Parse Input Input can be parsed into the object base in several ways. The easiest is to create the file before starting the system. When all the files are loaded, apply the **Parse-Input** rule. The system will ask for the name of the file, including the path. The system makes no assumptions concerning the location of the input, so the application specialist must enter the name and complete path exactly as he would if trying to refer to it at the Unix prompt. The path may be relative to the current directory (e.g., **./input-files** if the file is in the **input-files** subdirectory of the current directory) or it can be an absolute path (e.g., **~sys-name/domain1/inputs**). The case of the path

names and file names must match exactly with how it appears to the operating system; the system does no case conversion.

An alternate method of parsing input is to use the emacs editor. The application specialist can load the file into an emacs window and then use emacs commands to parse a buffer or parse a region. The final way is to use the REFINE window and directly type the input. The application specialist can enter (#> followed by the input and ended with a) and carriage return. Although this is a quick way to define a new application definition, it is not recommended because it is error-prone and the input is not saved into a file that can be edited if errors are made.

Once the input has been parsed, the application specialist must tell REFINE to go to the application definition. This is done using the **mcn** (make current node) command. After the input has been parsed, type **(mcn 'applic-def-name)** where applic-def-name is the name of the application definition. This should be done before a rule search is performed.

B.3.3.2 Edit An Object Objects already defined as part of an application definition can be modified without having to change and re-parse the input file. After the application specialist selects the edit object rule, the system will ask which object is to be edited (it will give the current object as the default value). It will then list all the attributes that can be modified, supplying the current value as a default value. If the attribute is a set or sequence, the interaction will be different. If the set or sequence is empty, it will ask if he wants to add another element. If the application specialist answers yes, it will ask for all the appropriate information for an element of that set or sequence. If the set or sequence already contains elements, the application specialist will have four options: add elements, delete elements, make the set/sequence empty, or finish editing. If adding an element to a sequence, the system will display the sequence and ask which index to use for the new element. This is not done for sets since the order of elements of a set is not important.

B.3.3.3 Add An Object If the application specialist wants to add an object instance to the application definition, he can add an object interactively by applying the

add object rule. The system will ask which application definition the new object should be part of, the type of object to be added (i.e., the object class name), the name of the new instance, and all of the attributes associated with objects of the specified class. If the object class name is not valid, or if the instance name is already used, the system will display an error message and discontinue this function.

B.3.3.4 Delete An Object The application specialist can remove objects from application definitions. After the application specialist applies the delete object rule, this function asks for the name of the object to be removed. It will then verify that the user really wants to delete the object. If the object is referenced by a subsystem or other object in the application definition, the application specialist is responsible for making modifications to the affected objects. If the changes are not made, the application definition will fail semantic checks.

B.3.3.5 Load Object The application specialist can select the load object rule to load additional object instances from the technology base into an application definition. The system will ask for the path name, display the objects available, and ask which one is to be loaded. It will then ask for an application definition name. If all the input is valid, it will load the object into the object base and assign it to the specified application definition.

B.3.3.6 Add Generic Instance Just as generic instances can be specified using the grammar, they can also be specified interactively by applying the add generic instance rule. The system prompts the application specialist for path of the generic object, displays the possible generic objects, asks for the name of the generic to use, asks for the application definition name, prompts for the name of the new instance to be created, and then asks for values for each of the parameters. This creates a generic instance, but does not instantiate it into a new object; this is done as part of the complete application definition function.

B.3.3.7 Complete Application Definition After the application specialist completes the application definition in the object base, or after he interactively adds a generic instance, he must then apply the complete application definition rule. Three different

types of input require further processing before the application definition is ready for further processing: load objects, incomplete objects, and generic instances. Load objects must be physically loaded from the technology base to the object base, incomplete objects must be completely defined, and generic instances must be instantiated. During this process, the system asks for the application name, and then searches for the three types of objects that require processing. If the system finds an error in one of the objects, it will give the user the opportunity to edit the object to fix the problem or to remove it from the application definition.

B.3.3.8 Save Object and Save Application Selecting the save object or save application option allows the application specialist to save specific object instances or application definitions to the technology base. The system will ask which object/application he wants to save and the path name. The path name specified is appended to the root directory for all saved objects/applications. If the object/application is to be saved in this root directory, the application specialist just hits return; if it is to be saved in a subdirectory, he must enter the name of the subdirectory. The system will ask the application specialist to verify the information before saving.

Saved objects and applications are stored in the exact same state as they exist in the object base. For example, if the import and export areas of a subsystem have already been built and the import sources have been identified, this information will be stored. If this information has not yet been defined, it will not be stored, but it is not an error. The application specialist may wish to save an application definition after a significant amount of processing has been done as a backup in case of a system failure.

The system does not have any predefined classification scheme, but objects and applications can be stored in separate subdirectories like generic objects. The difference here is that the application specialist is the one putting the objects into these subdirectories. The application specialist must work with the software engineer to establish these subdirectories before running the system.

B.3.3.9 Load Application A previously defined application can be reloaded into the object base. If the application specialist wants to return to an application definition from a previous session, this is the rule he applies. If the application specialist chooses to load a new application and one already exists in the object base, the system will print a message warning the user that he may create an inconsistent state. The system will ask for a path and display the applications that are available. It will then ask for the name of the file to be loaded. As with parsing new input files, the application specialist must tell REFINE to make the new application the current node as described in Section B.3.3.1.

The system can support several application definitions in the object base at one time, but this is potentially dangerous. If more than one application definition uses the same name for an object, only *one* object with that name will exist. The application specialist may be incorrectly using an object defined for a different application definition. The safest way to handle this situation is to save the current application definition, erase it, and then load the second application definition.

B.3.3.10 Erase Application Definition An entire application definition can be removed from the object base using the erase application definition rule. If the application definition was saved to a file, the file is *not* erased. This function is useful if the application specialist wishes to re-parse the input file or wants to load a new application definition.

B.4 Hints on Building Application Definitions

The application specialist has a lot of flexibility when creating application specifications. Some techniques work better in certain situations. It will take some experience with the system to figure out the best means of accomplishing different tasks.

B.4.1 Modifying the Object Base The current interface can be cumbersome to use at times, especially if the input requires substantial changes. If many changes are required, it may be easier for the application specialist to erase the application definition, fix the input file, and re-parse the input into the data base. If many changes have been made to

the import and export areas, it may be easier to edit interactively and then, during the semantic check, make the necessary changes to the affected import sources.

B.4.2 Creating New Application Definitions The application specialist can take several different approaches to building application definitions. Each new definition can be started from scratch and, until a new domain's technology base is established, this will be the only approach. If the software engineer has developed generic objects, or if objects have already been saved to the technology base, they can be used as high-level components.

B.4.2.1 Using Existing Objects in a New Application Definition If the application specialist wants to reuse parts of previous application definitions he can either copy the input from the file into a new file and the fill in any additional objects, or he can save the objects from the object base to the technology base and then use those exact objects again. The main difference in the two approaches is that the objects that are saved to the technology base might (depending on when they were saved) have some of the interconnections (the sources for the import objects) already defined. This information can be especially useful for a large subsystem.

When building the new application definition from existing objects in the technology base, the application specialist may elect to keep some of the defined connections; he only needs to make changes to connect the object to other objects in the application. However, if a subsystem is saved and then used in another application definition, it may still reference external sources for its imports. When the application specialist performs semantic checks, he will be asked if he wants to change the source for imports that already have sources. If the source is no longer defined in the application object, the application specialist must redefine the source. If this is not done, the execution will not run properly.

B.4.2.2 Building Up Components It is possible to slowly build up higher and higher level domain components. For example, a generic half-adder exists for the validating domain. A full adder can be built using two instances of the generic half-adder along with the appropriate interconnections. The full adder subsystem can be saved to the technology base for future use. If a full adder is needed for an application, all the required objects that

comprise the full adder can be loaded (using the grammar). The disadvantage of using existing objects (versus a generic or a copying input from another file) is that only one copy of each can be used per application definition.

Appendix C. *Validating Domain Code*

```
!! in-package("RU")
!! in-grammar('syntax)

#||
 File name: gram-logic.re

 Description: Grammar for the logic circuit domain (the validating domain)

 Rules:
  None

 Functions:
  None

||#


grammar Circuits

  no-patterns

  inherits-from OCU

  start-classes Spec-Obj, subsystem-obj, incomplete-obj, Generic-Obj,
    And-Gate-Obj, Or-Gate-Obj, Nand-Gate-Obj, Nor-Gate-Obj, Not-Gate-Obj,
    Switch-Obj, LED-Obj, JK-Flip-FLop-Obj


  file-classes Spec-Obj,  subsystem-obj, incomplete-obj, Generic-Obj,
    And-Gate-Obj, Or-Gate-Obj, Nand-Gate-Obj, Nor-Gate-Obj, Not-Gate-Obj,
    Switch-Obj, LED-Obj, JK-Flip-FLop-Obj


   productions


    And-Gate-Obj       ::= ["and-gate" name
                           {["delay:" and-gate-obj-delay]
                            [(["is mil-spec" !! and-gate-obj-mil-spec?] |
                              ["not mil-spec" ~!! and-gate-obj-mil-spec?])]
                            ["manufacturer:" and-gate-obj-manufacturer]
                            ["power level:" and-gate-obj-power-level] } ]
                                                    builds And-Gate-Obj,

    Or-Gate-Obj        ::= ["or-gate" name
                           {["delay:" or-gate-obj-delay]
                            [(["is mil-spec" !! or-gate-obj-mil-spec?] |
```

```
                    ["not mil-spec" ~!! or-gate-obj-mil-spec?])]
                  ["manufacturer:" or-gate-obj-manufacturer]
                  ["power level:" or-gate-obj-power-level] } ]
                                        builds Or-Gate-Obj,

Nand-Gate-Obj     ::= ["nand-gate" name
                     {["delay:" nand-gate-obj-delay]
                      [(["is mil-spec" !! nand-gate-obj-mil-spec?] |
                        ["not mil-spec" ~!! nand-gate-obj-mil-spec?])]
                      ["manufacturer:" nand-gate-obj-manufacturer]
                      ["power level:" nand-gate-obj-power-level] } ]
                                        builds Nand-Gate-Obj,

Nor-Gate-Obj      ::= ["nor-gate" name
                     {["delay:" nor-gate-obj-delay]
                      [(["mil-spec" !! nor-gate-obj-mil-spec?] |
                        ["not mil-spec" ~!! nor-gate-obj-mil-spec?])]
                      ["manufacturer:" nor-gate-obj-manufacturer]
                      ["power level:" nor-gate-obj-power-level] } ]
                                        builds Nor-Gate-Obj,

Not-Gate-Obj      ::= ["not-gate" name
                     {["delay:" not-gate-obj-delay]
                      [(["is mil-spec" !! not-gate-obj-mil-spec?] |
                        ["not mil-spec" ~!! not-gate-obj-mil-spec?])]
                      ["manufacturer:" not-gate-obj-manufacturer]
                      ["power level:" not-gate-obj-power-level] } ]
                                        builds Not-Gate-Obj,

JK-Flip-FLop-Obj  ::= ["jk-flip-flop" name
                     {["delay:" jk-flip-flop-obj-delay]
                      [(["is mil-spec" !! jk-flip-flop-obj-mil-spec?] |
                        ["not mil-spec" ~!! jk-flip-flop-obj-mil-spec?])]
                      ["manufacturer:" jk-flip-flop-obj-manufacturer]
                      ["power level:" jk-flip-flop-obj-power-level]
                      ["set-up delay:" jk-flip-flop-obj-set-up-delay]
                      ["hold delay:" jk-flip-flop-obj-hold-delay]
                      [(["state on" !! jk-flip-flop-obj-state]|
                        ["state off" ~!!jk-flip-flop-obj-state]) ] } ]
                                        builds JK-FLip-FLop-Obj,


Switch-Obj        ::= ["switch" name
                     {["delay:" switch-obj-delay]
                      [(["is debounced" !! switch-obj-debounced] |
                        ["not debounced" ~!! switch-obj-debounced])]
                      ["manufacturer:" switch-obj-manufacturer]
                      ["position:" switch-obj-position] } ]
                                        builds Switch-Obj,

LED-Obj           ::= ["led" name
```

```
                        {["manufacturer:" led-obj-manufacturer]
                         ["color:" led-obj-color] } ]   builds LED-Obj


  symbol-start-chars
     "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ./*"

  symbol-continue-chars
     "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ-0123456789./*?"

  comments "%" matching "
"

 precedence

    for expression brackets "(" matching ")"
      (same-level "and", "or" associativity left),
      (same-level "<", "<=", "=", ">=", ">", "/=" associativity none),
      (same-level "+","-"  associativity left),
      (same-level "*", "/", "mod"  associativity left),
      (same-level "not"  associativity none),
      (same-level "abs"  associativity none),
      (same-level "**" associativity right)

  end
```

```
!! in-package("RU")
!! in-grammar('user)


%%  File name:  and-gate.re

 var AND-GATE-OBJ       : object-class subtype-of Primitive-Obj

 var AND-GATE-OBJ-INPUT-DATA   : set(import-obj) =
       {set-attrs (make-object('import-obj),
                   'import-name, 'in1,
                   'import-category, 'signal,
                   'import-type-data, 'boolean),

       set-attrs (make-object('import-obj),
                   'import-name, 'in2,
                   'import-category, 'signal,
                   'import-type-data, 'boolean)}

 var AND-GATE-OBJ-OUTPUT-DATA  : set(export-obj) =
       {set-attrs (make-object('export-obj),
                   'export-name, 'out1,
                   'export-category, 'signal,
                   'export-type-data, 'boolean)}

 var AND-GATE-OBJ-COEFFICIENTS : map(AND-GATE-OBJ, set(name-value-obj))
  computed-using
    AND-GATE-OBJ-COEFFICIENTS(x) = {}


 var AND-GATE-OBJ-UPDATE-FUNCTION : map(AND-GATE-OBJ, symbol)
   computed-using
     AND-GATE-OBJ-UPDATE-FUNCTION(x) = 'AND-GATE-OBJ-UPDATE

% Other Attributes:
 var AND-GATE-OBJ-DELAY : map(AND-GATE-OBJ, integer)
   computed-using
    AND-GATE-OBJ-DELAY(x) = 0

 var AND-GATE-OBJ-MANUFACTURER :  map(AND-GATE-OBJ, string)
   computed-using
    AND-GATE-OBJ-MANUFACTURER(x) = " "

 var AND-GATE-OBJ-MIL-SPEC? :  map(AND-GATE-OBJ, boolean)
   computed-using
    AND-GATE-OBJ-MIL-SPEC?(x) = nil

 var AND-GATE-OBJ-POWER-LEVEL :  map(AND-GATE-OBJ, real)
   computed-using
    AND-GATE-OBJ-POWER-LEVEL(x) = 0.0
```

```
form Make-AND-GATE-Names-Unique
 unique-names-class('AND-GATE-OBJ, true)


function AND-GATE-OBJ-UPDATE (subsystem : subsystem-obj,
                              and-gate  : AND-GATE-OBJ) =

 format(debug-on, "AND-GATE-OBJ-UPDATE on ~s~%", name(and-gate));

 let (in1 : boolean = get-import('in1, subsystem, and-gate),
      in2 : boolean = get-import('in2, subsystem, and-gate))

 set-export(subsystem, and-gate, 'out1, in1 & in2)



function AND-GATE-OBJ-NEW-UPDATE (subsystem : subsystem-obj,
                                  and-gate  : AND-GATE-OBJ) =

 format(t, "AND-GATE-OBJ-NEW-UPDATE on ~s~%", name(and-gate))
```

```
!! in-package("RU")
!! in-grammar('user)


%% File name:  or-gate.re

 var OR-GATE-OBJ       : object-class subtype-of Primitive-Obj

 var OR-GATE-OBJ-INPUT-DATA   : set(import-obj) =
       {set-attrs (make-object('import-obj),
                   'import-name, 'in1,
                   'import-category, 'signal,
                   'import-type-data, 'boolean),

       set-attrs (make-object('import-obj),
                   'import-name, 'in2,
                   'import-category, 'signal,
                   'import-type-data, 'boolean)}

 var OR-GATE-OBJ-OUTPUT-DATA  : set(export-obj) =
       {set-attrs (make-object('export-obj),
                   'export-name, 'out1,
                   'export-category, 'signal,
                   'export-type-data, 'boolean)}

 var OR-GATE-OBJ-COEFFICIENTS : map(OR-GATE-OBJ, set(name-value-obj))
  computed-using
    OR-GATE-OBJ-COEFFICIENTS(x) = {}


 var OR-GATE-OBJ-UPDATE-FUNCTION : map(OR-GATE-OBJ, symbol)
   computed-using
     OR-GATE-OBJ-UPDATE-FUNCTION(x) = 'OR-GATE-OBJ-UPDATE

% Other Attributes:
 var OR-GATE-OBJ-DELAY : map(OR-GATE-OBJ, integer)
   computed-using
    OR-GATE-OBJ-DELAY(x) = 0

 var OR-GATE-OBJ-MANUFACTURER :  map(OR-GATE-OBJ, string)
   computed-using
    OR-GATE-OBJ-MANUFACTURER(x) = " "

 var OR-GATE-OBJ-MIL-SPEC? :  map(OR-GATE-OBJ, boolean)
   computed-using
    OR-GATE-OBJ-MIL-SPEC?(x) = nil

 var OR-GATE-OBJ-POWER-LEVEL :  map(OR-GATE-OBJ, real)
   computed-using
    OR-GATE-OBJ-POWER-LEVEL(x) = 0.0
```

```
form Make-OR-GATE-Names-Unique
 unique-names-class('OR-GATE-OBJ, true)


function OR-GATE-OBJ-UPDATE (subsystem : subsystem-obj,
                             or-gate  : OR-GATE-OBJ) =

 format(debug-on, "OR-GATE-OBJ-UPDATE on ~s~%", name(or-gate));

 let (in1 : boolean = get-import('in1, subsystem, or-gate),
      in2 : boolean = get-import('in2, subsystem, or-gate))

 set-export(subsystem, or-gate, 'out1, (in1 or in2))



function OR-GATE-OBJ-NEW-UPDATE (subsystem : subsystem-obj,
                                 or-gate  : OR-GATE-OBJ) =

 format(t, "OR-GATE-OBJ-NEW-UPDATE on ~s~%", name(or-gate))
```

```
!! in-package("RU")
!! in-grammar('user)


%%  File name:  nand-gate.re

 var NAND-GATE-OBJ       : object-class subtype-of Primitive-Obj

 var NAND-GATE-OBJ-INPUT-DATA   : set(import-obj) =
       {set-attrs (make-object('import-obj),
                   'import-name, 'in1,
                   'import-category, 'signal,
                   'import-type-data, 'boolean),

       set-attrs (make-object('import-obj),
                   'import-name, 'in2,
                   'import-category, 'signal,
                   'import-type-data, 'boolean)}

 var NAND-GATE-OBJ-OUTPUT-DATA  : set(export-obj) =
       {set-attrs (make-object('export-obj),
                   'export-name, 'out1,
                   'export-category, 'signal,
                   'export-type-data, 'boolean)}

 var NAND-GATE-OBJ-COEFFICIENTS : map(NAND-GATE-OBJ, set(name-value-obj))
  computed-using
    NAND-GATE-OBJ-COEFFICIENTS(x) = {}


 var NAND-GATE-OBJ-UPDATE-FUNCTION : map(NAND-GATE-OBJ, symbol)
   computed-using
     NAND-GATE-OBJ-UPDATE-FUNCTION(x) = 'NAND-GATE-OBJ-UPDATE

% Other Attributes:
 var NAND-GATE-OBJ-DELAY : map(NAND-GATE-OBJ, integer)
   computed-using
    NAND-GATE-OBJ-DELAY(x) = 0

 var NAND-GATE-OBJ-MANUFACTURER :  map(NAND-GATE-OBJ, string)
   computed-using
    NAND-GATE-OBJ-MANUFACTURER(x) = " "

 var NAND-GATE-OBJ-MIL-SPEC? :  map(NAND-GATE-OBJ, boolean)
   computed-using
    NAND-GATE-OBJ-MIL-SPEC?(x) = nil

 var NAND-GATE-OBJ-POWER-LEVEL :  map(NAND-GATE-OBJ, real)
   computed-using
    NAND-GATE-OBJ-POWER-LEVEL(x) = 0.0
```

```
form Make-NAND-GATE-Names-Unique
 unique-names-class('NAND-GATE-OBJ, true)


function NAND-GATE-OBJ-UPDATE (subsystem  : subsystem-obj,
                               nand-gate  : NAND-GATE-OBJ) =

 format(debug-on, "NAND-GATE-OBJ-UPDATE on ~s~%", name(nand-gate));

 let (in1 : boolean = get-import('in1, subsystem, nand-gate),
      in2 : boolean = get-import('in2, subsystem, nand-gate))

 set-export(subsystem, nand-gate, 'out1, ~(in1 & in2))



function NAND-GATE-OBJ-NEW-UPDATE (subsystem  : subsystem-obj,
                                   nand-gate  : NAND-GATE-OBJ) =

 format(t, "NAND-GATE-OBJ-NEW-UPDATE on ~s~%", name(nand-gate))
```

```
!! in-package("RU")
!! in-grammar('user)


%%  File name:  nor-gate.re

 var NOR-GATE-OBJ       : object-class subtype-of Primitive-Obj

 var NOR-GATE-OBJ-INPUT-DATA   : set(import-obj) =
       {set-attrs (make-object('import-obj),
                   'import-name, 'in1,
                   'import-category, 'signal,
                   'import-type-data, 'boolean),

       set-attrs (make-object('import-obj),
                   'import-name, 'in2,
                   'import-category, 'signal,
                   'import-type-data, 'boolean)}

 var NOR-GATE-OBJ-OUTPUT-DATA  : set(export-obj) =
       {set-attrs (make-object('export-obj),
                   'export-name, 'out1,
                   'export-category, 'signal,
                   'export-type-data, 'boolean)}

 var NOR-GATE-OBJ-COEFFICIENTS : map(NOR-GATE-OBJ, set(name-value-obj))
  computed-using
    NOR-GATE-OBJ-COEFFICIENTS(x) = {}


 var NOR-GATE-OBJ-UPDATE-FUNCTION : map(NOR-GATE-OBJ, symbol)
   computed-using
     NOR-GATE-OBJ-UPDATE-FUNCTION(x) = 'NOR-GATE-OBJ-UPDATE

% Other Attributes:
 var NOR-GATE-OBJ-DELAY : map(NOR-GATE-OBJ, integer)
   computed-using
    NOR-GATE-OBJ-DELAY(x) = 0

 var NOR-GATE-OBJ-MANUFACTURER :  map(NOR-GATE-OBJ, string)
   computed-using
    NOR-GATE-OBJ-MANUFACTURER(x) = " "

 var NOR-GATE-OBJ-MIL-SPEC? :  map(NOR-GATE-OBJ, boolean)
   computed-using
    NOR-GATE-OBJ-MIL-SPEC?(x) = nil

 var NOR-GATE-OBJ-POWER-LEVEL :  map(NOR-GATE-OBJ, real)
   computed-using
    NOR-GATE-OBJ-POWER-LEVEL(x) = 0.0
```

```
form Make-NOR-GATE-Names-Unique
 unique-names-class('NOR-GATE-OBJ, true)


function NOR-GATE-OBJ-UPDATE (subsystem : subsystem-obj,
                              nor-gate  : NOR-GATE-OBJ) =


 format(debug-on, "NOR-GATE-OBJ-UPDATE on ~s~%", name(nor-gate));

 let (in1 : boolean = get-import('in1, subsystem, nor-gate),
      in2 : boolean = get-import('in2, subsystem, nor-gate))

 set-export(subsystem, nor-gate, 'out1, ~(in1 or in2))




function NOR-GATE-OBJ-NEW-UPDATE (subsystem : subsystem-obj,
                                  nor-gate  : NOR-GATE-OBJ) =

 format(t, "NOR-GATE-OBJ-NEW-UPDATE on ~s~%", name(nor-gate))
```

```
!! in-package("RU")
!! in-grammar('user)


%%  File name:  jk-flip-flop.re

 var JK-FLIP-FLOP-OBJ       : object-class subtype-of Primitive-Obj

 var JK-FLIP-FLOP-OBJ-INPUT-DATA   : set(import-obj) =
       {set-attrs (make-object('import-obj),
                   'import-name, 'J,
                   'import-category, 'signal,
                   'import-type-data, 'boolean),
       set-attrs (make-object('import-obj),
                   'import-name, 'K,
                   'import-category, 'signal,
                   'import-type-data, 'boolean),
       set-attrs (make-object('import-obj),
                   'import-name, 'Clk,
                   'import-category, 'signal,
                   'import-type-data, 'boolean)}

 var JK-FLIP-FLOP-OBJ-OUTPUT-DATA  : set(export-obj) =
       {set-attrs (make-object('export-obj),
                   'export-name, 'Q,
                   'export-category, 'signal,
                   'export-type-data, 'boolean),
        set-attrs (make-object('export-obj),
                   'export-name, 'Q-Bar,
                   'export-category, 'signal,
                   'export-type-data, 'boolean)}

 var JK-FLIP-FLOP-OBJ-COEFFICIENTS : map(JK-FLIP-FLOP-OBJ, set(name-value-obj))
  computed-using
    JK-FLIP-FLOP-OBJ-COEFFICIENTS(x) = {}


 var JK-FLIP-FLOP-OBJ-UPDATE-FUNCTION : map(JK-FLIP-FLOP-OBJ, symbol)
   computed-using
     JK-FLIP-FLOP-OBJ-UPDATE-FUNCTION(x) = 'JK-FLIP-FLOP-OBJ-UPDATE

% Other Attributes:
 var JK-FLIP-FLOP-OBJ-DELAY : map(JK-FLIP-FLOP-OBJ, integer)
   computed-using
    JK-FLIP-FLOP-OBJ-DELAY(x) = 0

 var JK-FLIP-FLOP-OBJ-MANUFACTURER :  map(JK-FLIP-FLOP-OBJ, string)
   computed-using
    JK-FLIP-FLOP-OBJ-MANUFACTURER(x) = " "

 var JK-FLIP-FLOP-OBJ-MIL-SPEC? :  map(JK-FLIP-FLOP-OBJ, boolean)
```

```
  computed-using
   JK-FLIP-FLOP-OBJ-MIL-SPEC?(x) = nil

 var JK-FLIP-FLOP-OBJ-POWER-LEVEL :  map(JK-FLIP-FLOP-OBJ, real)
  computed-using
   JK-FLIP-FLOP-OBJ-POWER-LEVEL(x) = 0.0

 var JK-FLIP-FLOP-OBJ-SET-UP-DELAY :  map(JK-FLIP-FLOP-OBJ, integer)
  computed-using
   JK-FLIP-FLOP-OBJ-SET-UP-DELAY(x) = 0

 var JK-FLIP-FLOP-OBJ-HOLD-DELAY :  map(JK-FLIP-FLOP-OBJ, real)
  computed-using
   JK-FLIP-FLOP-OBJ-HOLD-DELAY(x) = 0.0

 var JK-FLIP-FLOP-OBJ-STATE :  map(JK-FLIP-FLOP-OBJ, boolean)
  computed-using
   JK-FLIP-FLOP-OBJ-STATE(x) = nil



 form Make-JK-FLIP-FLOP-Names-Unique
  unique-names-class('JK-FLIP-FLOP-OBJ, true)



function JK-FLIP-FLOP-OBJ-UPDATE (subsystem    : subsystem-obj,
                                  jk-flip-flop : JK-FLIP-FLOP-OBJ) =

 format(debug-on, "JK-FLIP-FLOP-OBJ-UPDATE on ~s~%", name(jk-flip-flop));

 let (j   : boolean = get-import('J, subsystem, jk-flip-flop),
      k   : boolean = get-import('K, subsystem, jk-flip-flop),
      clk : boolean = get-import('Clk, subsystem, jk-flip-flop))

 (if ~j & k & clk then
    JK-FLIP-FLOP-OBJ-STATE(jk-flip-flop) <- nil

 elseif j & k & clk then
    JK-FLIP-FLOP-OBJ-STATE(jk-flip-flop) <- ~JK-FLIP-FLOP-OBJ-STATE(jk-flip-flop)

 elseif j & ~k and clk then
    JK-FLIP-FLOP-OBJ-STATE(jk-flip-flop) <- true
 );

 set-export(subsystem, jk-flip-flop, 'Q, JK-FLIP-FLOP-OBJ-STATE(jk-flip-flop));
 set-export(subsystem, jk-flip-flop, 'Q-Bar,
                     ~JK-FLIP-FLOP-OBJ-STATE(jk-flip-flop))
```

```
function JK-FLIP-FLOP-OBJ-NEW-UPDATE (subsystem     : subsystem-obj,
                                       jk-flip-flop : JK-FLIP-FLOP-OBJ) =

 format(t, "JK-FLIP-FLOP-OBJ-NEW-UPDATE on ~s~%", name(jk-flip-flop))
```

```
!! in-package("RU")
!! in-grammar('user)


%% File name:  not-gate.re

 var NOT-GATE-OBJ       : object-class subtype-of Primitive-Obj

 var NOT-GATE-OBJ-INPUT-DATA   : set(import-obj) =
       {set-attrs (make-object('import-obj),
                   'import-name, 'in1,
                   'import-category, 'signal,
                   'import-type-data, 'boolean)}

 var NOT-GATE-OBJ-OUTPUT-DATA  : set(export-obj) =
       {set-attrs (make-object('export-obj),
                   'export-name, 'out1,
                   'export-category, 'signal,
                   'export-type-data, 'boolean)}

 var NOT-GATE-OBJ-COEFFICIENTS : map(NOT-GATE-OBJ, set(name-value-obj))
  computed-using
    NOT-GATE-OBJ-COEFFICIENTS(x) = {}


 var NOT-GATE-OBJ-UPDATE-FUNCTION : map(NOT-GATE-OBJ, symbol)
   computed-using
     NOT-GATE-OBJ-UPDATE-FUNCTION(x) = 'NOT-GATE-OBJ-UPDATE

% Other Attributes:
 var NOT-GATE-OBJ-DELAY : map(NOT-GATE-OBJ, integer)
   computed-using
    NOT-GATE-OBJ-DELAY(x) = 0

 var NOT-GATE-OBJ-MANUFACTURER :  map(NOT-GATE-OBJ, string)
   computed-using
    NOT-GATE-OBJ-MANUFACTURER(x) = " "

 var NOT-GATE-OBJ-MIL-SPEC? :  map(NOT-GATE-OBJ, boolean)
   computed-using
    NOT-GATE-OBJ-MIL-SPEC?(x) = nil

 var NOT-GATE-OBJ-POWER-LEVEL :  map(NOT-GATE-OBJ, real)
   computed-using
    NOT-GATE-OBJ-POWER-LEVEL(x) = 0.0

 form Make-NOT-GATE-Names-Unique
  unique-names-class('NOT-GATE-OBJ, true)
```

```
function NOT-GATE-OBJ-UPDATE (subsystem : subsystem-obj,
                              not-gate  : NOT-GATE-OBJ) =

 format(debug-on, "NOT-GATE-OBJ-UPDATE on ~s~%", name(not-gate));

 let (in1 : boolean = get-import('in1, subsystem, not-gate))

 set-export(subsystem, not-gate, 'out1, ~(in1))




function NOT-GATE-OBJ-NEW-UPDATE (subsystem : subsystem-obj,
                                  not-gate  : NOT-GATE-OBJ) =

 format(t, "NOT-GATE-OBJ-NEW-UPDATE on ~s~%", name(not-gate))
```

```
!! in-package("RU")
!! in-grammar('user)


%% File name:  led.re

 var LED-OBJ       : object-class subtype-of Primitive-Obj

 var LED-OBJ-INPUT-DATA   : set(import-obj) =
       {set-attrs (make-object('import-obj),
                       'import-name, 'in1,
                       'import-category, 'signal,
                       'import-type-data, 'boolean)}

 var LED-OBJ-OUTPUT-DATA  : set(export-obj) = {}

 var LED-OBJ-COEFFICIENTS : map(LED-OBJ, set(name-value-obj))
  computed-using
    LED-OBJ-COEFFICIENTS(x) = {}


 var LED-OBJ-UPDATE-FUNCTION : map(LED-OBJ, symbol)
   computed-using
     LED-OBJ-UPDATE-FUNCTION(x) = 'LED-OBJ-UPDATE

% Other Attributes:
 var LED-OBJ-MANUFACTURER :  map(LED-OBJ, string)
   computed-using
    LED-OBJ-MANUFACTURER(x) = " "

 var LED-OBJ-COLOR :  map(LED-OBJ, symbol)
   computed-using
    LED-OBJ-COLOR(x) = 'red

 form Make-LED-Names-Unique
  unique-names-class('LED-OBJ, true)



function LED-OBJ-UPDATE (subsystem : subsystem-obj,
                                led  : LED-OBJ) =

 format(t, "LED-OBJ-UPDATE on ~s~%", name(led));

 let (display-value : symbol = 'false)

 (if get-import('in1, subsystem, led) then
    display-value <- 'true
 );
 format(t, "LED ~s = ~s~%", name(led), display-value)
```

```
function LED-OBJ-NEW-UPDATE (subsystem : subsystem-obj,
                             led   : LED-OBJ) =

 format(t, "LED-OBJ-NEW-UPDATE on ~s~%", name(led))
```

```
!! in-package("RU")
!! in-grammar('user)


%%  File name:  switch.re

 var SWITCH-OBJ       : object-class subtype-of Primitive-Obj

 var SWITCH-OBJ-INPUT-DATA   : set(import-obj) =  {}

 var SWITCH-OBJ-OUTPUT-DATA  : set(export-obj) =
       {set-attrs (make-object('export-obj),
                   'export-name, 'out1,
                   'export-category, 'signal,
                   'export-type-data, 'boolean)}

 var SWITCH-OBJ-COEFFICIENTS : map(SWITCH-OBJ, set(name-value-obj))
  computed-using
    SWITCH-OBJ-COEFFICIENTS(x) = {}


 var SWITCH-OBJ-UPDATE-FUNCTION : map(SWITCH-OBJ, symbol)
   computed-using
     SWITCH-OBJ-UPDATE-FUNCTION(x) = 'SWITCH-OBJ-UPDATE

% Other Attributes:
 var SWITCH-OBJ-MANUFACTURER :  map(SWITCH-OBJ, string)
   computed-using
    SWITCH-OBJ-MANUFACTURER(x) = " "

 var SWITCH-OBJ-DEBOUNCED :  map(SWITCH-OBJ, boolean)
   computed-using
    SWITCH-OBJ-DEBOUNCED(x) = nil

 var SWITCH-OBJ-DELAY :  map(SWITCH-OBJ, integer)
   computed-using
    SWITCH-OBJ-DELAY(x) = 0

 var SWITCH-OBJ-POSITION :  map(SWITCH-OBJ, symbol)
   computed-using
    SWITCH-OBJ-POSITION(x) = 'on

 form Make-SWITCH-Names-Unique
  unique-names-class('SWITCH-OBJ, true)



function SWITCH-OBJ-UPDATE (subsystem : subsystem-obj,
                            switch    : SWITCH-OBJ) =

 format(debug-on, "SWITCH-GATE-OBJ-UPDATE on ~s~%", name(switch));
```

```
let (signal : boolean = false)

(if SWITCH-OBJ-POSITION(switch) = 'ON then
  signal <- true
);

format(t, "Switch position = ~s~%", SWITCH-OBJ-POSITION(switch));

set-export(subsystem, switch, 'out1, signal)




function SWITCH-OBJ-NEW-UPDATE (subsystem : subsystem-obj,
                                switch   : SWITCH-OBJ) =

format(t, "SWITCH-OBJ-NEW-UPDATE on ~s~%", name(switch))
```

Appendix D. *Sample Session*

This appendix shows several sample sessions to demonstrate how this portion of the system works. First, we built a universal left-right presettable shift register and then a binary array multiplier. Most of the functions available to the application specialist are demonstrated in these two sessions, the ones not shown are obvious as to their functions. Several comments were added to the session outputs to indicate the action being taken, these comments will be inclosed between "(*" and "*)" so that they will not be confused with normal input and output.

D.1 Universal Shift Register

We built a universal left-right presettable shift register shown in Figure D.1 in two steps: we built a subsystem that defines the shift register, saved it, and then used that shift register subsystem as part of another application.

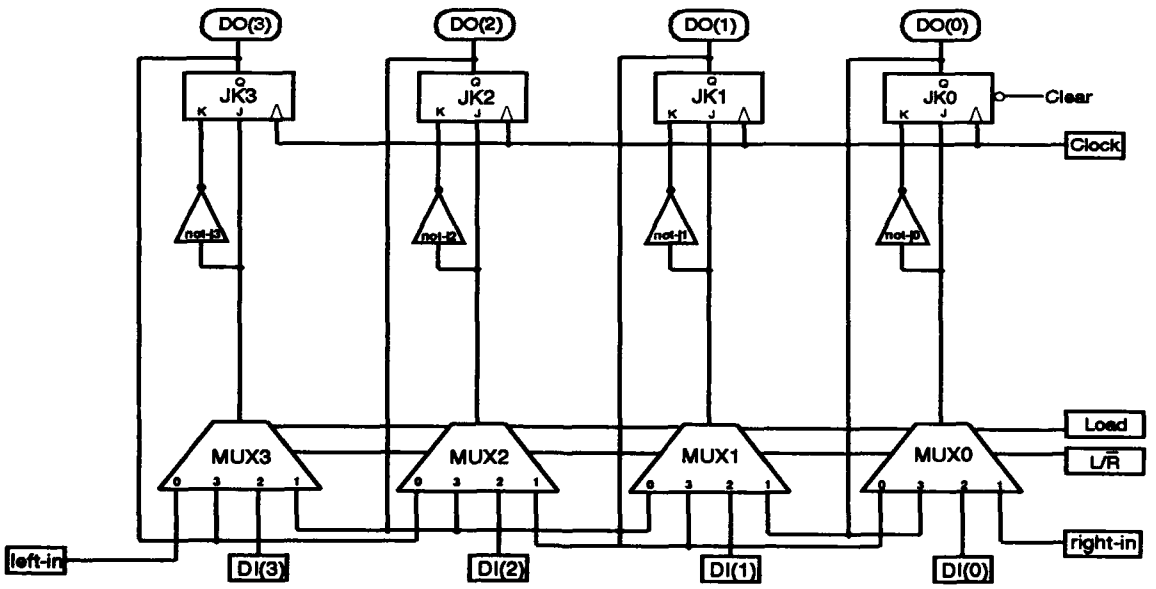

Figure D.1. Universal Shift Register

The code for building the subsystem was as follows:

```
application definition lr-shr-applic
```

```
jk-flip-flop jk1 state off
jk-flip-flop jk2 state off
jk-flip-flop jk3 state off
jk-flip-flop jk4 state off

% use nots to make jk flip flop simulate a D flip flop
not-gate notj1
not-gate notj2
not-gate notj3
not-gate notj4

mux mux1
mux mux2
mux mux3
mux mux4

switch clock    position: on
switch load-switch     position: on
switch lr-bar   position: on % when on, shifts left
switch clear    position: off
switch left-in  position: off
switch right-in position:on

switch DI0 position: on
switch DI1 position: off
switch DI2 position: on
switch DI3 position: off

led DO0
led DO1
led DO2
led DO3

subsystem lr-shift-reg is
  controls:  jk1, notj1, jk2, notj2, jk3, notj3, jk4, notj4,
             mux1, mux2, mux3, mux4
  update procedure:
   update mux1
   update mux2
   update mux3
   update mux4

   update notj1
   update jk1
   update notj2
   update jk2
   update notj3
   update jk3
   update notj4
   update jk4
```

```
subsystem drive-shift is
  controls: lr-shift-reg, clock, load-switch, lr-bar, clear, left-in, right-in,
            DI0, DI1, DI2, DI3, DO3, DO2, DO1, DO0
   update procedure:
    update clear
    update clock
    update load-switch
    update lr-bar
    update left-in
    update right-in
    update DI3
    update DI2
    update DI1
    update DI0
    update lr-shift-reg
    update DO3
    update DO2
    update DO1
    update DO0

application do-shift is
  controls: drive-shift
    update procedure:
     update drive-shift
```

Although the additional subsystem, inputs (switches), and outputs (leds) are not necessarily required, they were used to test the subsystem and define the sources for the imports before it was saved. The session used to build and save the subsystem appears belos (user inputs follows the REFINE prompt .>):

```
(* the current node was set to the variable FATAL-ERROR since no input
has been parsed *)
.> (rs)
- Rules for: ##r USER var FATAL-ERROR: boolean = false -
1) ERASE-APPLICATION-DEFINITION
2) EDIT-AN-OBJECT
3) ADD-AN-OBJECT
4) DELETE-AN-OBJECT
5) SAVE-OBJECT
6) SAVE-APPLICATION
7) LOAD-OBJECT
8) LOAD-APPLICATION
9) PARSE-INPUT
```

```
10) ADD-GENERIC-INSTANCE
11) BUILD-GENERIC
12) SAVE-GENERIC
13) COMPLETE-APPLICATION-DEFINITION
15) CHECK-SEMANTICS
.> (ar 9)          (* parse in the input file *)
Enter the full name of the file: ./DSL/tests/demo2
##r USER var FATAL-ERROR: boolean = false
.> (mcn 'lr-shr-applic)
application definition LR-SHR-APPLIC
  JK1 JK2 JK3 JK4 NOTJ1 NOTJ2 NOTJ3 NOTJ4 MUX1 MUX2 MUX3 MUX4
    CLOCK LOAD-SWITCH LR-BAR CLEAR LEFT-IN RIGHT-IN DI0 DI1 DI2
    DI3 DO0 DO1 DO2 DO3 LR-SHIFT-REG DRIVE-SHIFT DO-SHIFT
.> (rs)
- Rules for: application definition LR-SHR-APPLIC
  JK1 JK2 JK3 JK4 NOTJ1 NOTJ2 NOTJ3 NOTJ4 MUX1 MUX2 MUX3 MUX4
    CLOCK LOAD-SWITCH LR-BAR CLEAR LEFT-IN RIGHT-IN DI0 DI1 DI2
    DI3 DO0 DO1 DO2 DO3 LR-SHIFT-REG DRIVE-SHIFT DO-SHIFT -
1) ERASE-APPLICATION-DEFINITION
2) EDIT-AN-OBJECT
3) ADD-AN-OBJECT
4) DELETE-AN-OBJECT
5) SAVE-OBJECT
6) SAVE-APPLICATION
7) LOAD-OBJECT
8) LOAD-APPLICATION
9) PARSE-INPUT
10) ADD-GENERIC-INSTANCE
11) BUILD-GENERIC
12) SAVE-GENERIC
13) COMPLETE-APPLICATION-DEFINITION
15) CHECK-SEMANTICS
.> (ar 15)  (* check the semantics and determine the sources for imports *)

More than one export can provide the data for S1
                    which is used by object MUX4
                                in subsystem LR-SHIFT-REG
Choose the export item (subsystem and component)
  that you wish to be the source of this data:
     1> subsystem "LR-SHIFT-REG" component "JK1" name "Q"
     2> subsystem "LR-SHIFT-REG" component "JK1" name "Q-BAR"
     ...
Enter the number corresponding to the source you want to use

(* defined all of the import sources (i.e., interconnections) *)
(* execute to test behavior *)

.> (ar 5)
Enter object to save:  (LR-SHR-APPLIC): lr-shift-reg
Enter the path (include last /)./objs/: univ-shr/
```

```
Do you really want to save ./objs/UNIV-SHR/LR-SHIFT-REG-SAVED? (y or n) y

DO you want to save the objects it controls also? (y or n) y
(* saves all of the objects controlled by the subsystem with the subsystem
   so they do not have to be loaded separately when it is used in a new
   application definition *)

Rule successfully applied.
application definition LR-SHR-APPLIC
  JK1 JK2 JK3 JK4 NOTJ1 NOTJ2 NOTJ3 NOTJ4 MUX1 MUX2 MUX3 MUX4
    CLOCK LOAD-SWITCH LR-BAR CLEAR LEFT-IN RIGHT-IN DI0 DI1 DI2
    DI3 DO0 DO1 DO2 DO3 LR-SHIFT-REG DRIVE-SHIFT DO-SHIFT
```

Now that we have the shift register saved, it can be used in the following application definition. Note that the objects used within the shift register are not redefined in this application definition; they are instead stored with the shift register. This demonstrates how a subsystem that was built as part of one application definition can be used in another.

```
application definition demo

object switch-obj, clock

switch load-switch  position: on
switch lr-bar       position: off  % when on, shifts left
switch clear        position: off
switch left-in      position: off
switch right-in     position:on

switch DI0 position: on
switch DI1 position: off
switch DI2 position: on
switch DI3 position: off

led DO0
led DO1
led DO2
led DO3

load univ-shr/lr-shift-reg

subsystem drive-shift is
  controls: lr-shift-reg, clock, load-switch, lr-bar, clear, left-in, right-in,
            DI0, DI1, DI2, DI3, DO3, DO2, DO1, DO0
   update procedure:
    % first load the input into the register
```

```
setstate load-switch (position, on)
setstate lr-bar (position, off)

update clear
update clock
update load-switch
update lr-bar
update left-in
update right-in
update DI3
update DI2
update DI1
update DI0
update lr-shift-reg
update DO3
update DO2
update DO1
update DO0

% now shift right - using original input still
setstate load-switch (position, off)
setstate lr-bar (position, off)
update load-switch
update lr-bar

update lr-shift-reg
update DO3
update DO2
update DO1
update DO0

update lr-shift-reg
update DO3
update DO2
update DO1
update DO0

update lr-shift-reg
update DO3
update DO2
update DO1
update DO0


update lr-shift-reg
update DO3
update DO2
update DO1
update DO0

% clear out the old stuff
```

```
setstate load-switch (position, on)
setstate lr-bar (position, off)
update clear
update clock
update load-switch
update lr-bar
update left-in
update right-in
update DI3
update DI2
update DI1
update DI0
update lr-shift-reg
update DO3
update DO2
update DO1
update DO0


% now shift left - using original input
 setstate load-switch (position, off)
 setstate lr-bar (position, on)
 update load-switch
 update lr-bar

 update DI3
 update DI2
 update DI1
 update DI0
 update lr-shift-reg
 update DO3
 update DO2
 update DO1
 update DO0

 update lr-shift-reg
 update DO3
 update DO2
 update DO1
 update DO0

 update lr-shift-reg
 update DO3
 update DO2
 update DO1
 update DO0

 update lr-shift-reg
 update DO3
 update DO2
 update DO1
```

```
    update DO0


application do-shift is
  controls: drive-shift
    update procedure:
     update drive-shift
```

The following shows how this application was parsed into the object base, completely defined, and then modified. It demonstrates more of the obect base manipulation functions available to the user.

```
.> (rs)
- Rules for: ##r USER var FATAL-ERROR: boolean = false -
1) ERASE-APPLICATION-DEFINITION
2) EDIT-AN-OBJECT
3) ADD-AN-OBJECT
4) DELETE-AN-OBJECT
5) SAVE-OBJECT
6) SAVE-APPLICATION
7) LOAD-OBJECT
8) LOAD-APPLICATION
9) PARSE-INPUT
10) ADD-GENERIC-INSTANCE
11) BUILD-GENERIC
12) SAVE-GENERIC
13) COMPLETE-APPLICATION-DEFINITION
15) CHECK-SEMANTICS
.> (ar 9) (* parse in the input file *)
Enter the full name of the file: ./DSL/tests/demo
Rule successfully applied.
##r USER var FATAL-ERROR: boolean = false

.> (mcn 'demo)
application definition DEMO
  CLOCK LOAD-SWITCH LR-BAR CLEAR LEFT-IN RIGHT-IN DIO DI1 DI2
    DI3 DO0 DO1 DO2 DO3 load UNIV-SHR/LR-SHIFT-REG DRIVE-SHIFT
    DO-SHIFT

.> (ar 13)                    (* the clock object is not fully defined,
                                  it is an incomplete object *)
Enter the application name:  (DEMO):
completing an incomplete object
Complete the information for object CLOCK:
Enter UPDATE-FUNCTION:  (SWITCH-OBJ-UPDATE1):
Enter MANUFACTURER ( ):
```

```
Enter DEBOUNCED (T/t for true, F/f for false:)  (NIL): t
Enter DELAY: (0): 2
Enter POSITION:  (ON): off
Rule successfully applied.
application definition DEMO
  LOAD-SWITCH LR-BAR CLEAR LEFT-IN RIGHT-IN DI0 DI1 DI2 DI3
    DO0 DO1 DO2 DO3 DRIVE-SHIFT DO-SHIFT CLOCK LR-SHIFT-REG JK1
    NOTJ1 JK2 NOTJ2 JK3 NOTJ3 JK4 NOTJ4 MUX1 MUX2 MUX3 MUX4

.> (pup 'clock)           (* now there is a new switch object called clock *)
CLOCK - the switch-obj
   re::class:  switch-obj
   parent-expr: #1<DEMO - the spec-obj>
   name:  clock
   switch-obj-update-function:  switch-obj-update1
   switch-obj-manufacturer: " "
   switch-obj-debounced: true
   switch-obj-delay: 2
   switch-obj-position:  off

.> (ar 2)                                    (* editing an object *)
Enter the name of the object to edit:  (DEMO): clock
Enter UPDATE-FUNCTION:  (SWITCH-OBJ-UPDATE1):
Enter MANUFACTURER ( ):
Enter DEBOUNCED (T/t for true, F/f for false:)  (T):
Enter DELAY: (2):
Enter POSITION:  (OFF): on
Rule successfully applied.
application definition DEMO
  LOAD-SWITCH LR-BAR CLEAR LEFT-IN RIGHT-IN DI0 DI1 DI2 DI3
    DO0 DO1 DO2 DO3 DRIVE-SHIFT DO-SHIFT CLOCK LR-SHIFT-REG JK1
    NOTJ1 JK2 NOTJ2 JK3 NOTJ3 JK4 NOTJ4 MUX1 MUX2 MUX3 MUX4

.> (ar 3)                                  (* adding a new object *)
Enter the application name:  (DEMO):
What type of object do you want to create?: and-gate-obj
What is the object's name?: new-and
Enter UPDATE-FUNCTION:  (AND-GATE-OBJ-UPDATE1):
Enter DELAY: (0): 1
Enter MANUFACTURER ( ):
Enter MIL-SPEC? (T/t for true, F/f for false:)  (NIL): f
Enter POWER-LEVEL:  (0.0): 1.2
Rule successfully applied.
application definition DEMO
  LOAD-SWITCH LR-BAR CLEAR LEFT-IN RIGHT-IN DI0 DI1 DI2 DI3
    DO0 DO1 DO2 DO3 DRIVE-SHIFT DO-SHIFT CLOCK LR-SHIFT-REG JK1
    NOTJ1 JK2 NOTJ2 JK3 NOTJ3 JK4 NOTJ4 MUX1 MUX2 MUX3 MUX4
    NEW-AND
.> (pup 'new-and)
NEW-AND - the and-gate-obj
   re::class:  and-gate-obj
```

```
   parent-expr: #1<DEMO - the spec-obj>
   and-gate-obj-update-function:  and-gate-obj-update1
   and-gate-obj-delay: 1
   and-gate-obj-manufacturer: " "
   and-gate-obj-mil-spec?: false
   and-gate-obj-power-level: 1.2
   name:  new-and

.> (ar 4)                                    (* deleting an object *)
What object do you want to delete?:  (DEMO): new-and

Are you sure you want to detete NEW-AND (y or n) y
Rule successfully applied.
application definition DEMO
  LOAD-SWITCH LR-BAR CLEAR LEFT-IN RIGHT-IN DIO DI1 DI2 DI3
    DOO DO1 DO2 DO3 DRIVE-SHIFT DO-SHIFT CLOCK LR-SHIFT-REG JK1
    NOTJ1 JK2 NOTJ2 JK3 NOTJ3 JK4 NOTJ4 MUX1 MUX2 MUX3 MUX4

.> (ar 5)                                    (* saving an object *)
Enter object to save:  (DEMO): load-switch
Enter the path (include last /)./objs/:

Do you really want to save ./objs/LOAD-SWITCH-SAVED? (y or n) y
Rule successfully applied.
application definition DEMO
  LOAD-SWITCH LR-BAR CLEAR LEFT-IN RIGHT-IN DIO DI1 DI2 DI3
    DOO DO1 DO2 DO3 DRIVE-SHIFT DO-SHIFT CLOCK LR-SHIFT-REG JK1
    NOTJ1 JK2 NOTJ2 JK3 NOTJ3 JK4 NOTJ4 MUX1 MUX2 MUX3 MUX4

.> (ar 4)
What object do you want to delete?:  (DEMO): load-switch

Are you sure you want to detete LOAD-SWITCH (y or n) y
Rule successfully applied.
application definition DEMO
  LR-BAR CLEAR LEFT-IN RIGHT-IN DIO DI1 DI2 DI3 DOO DO1 DO2
    DO3 DRIVE-SHIFT DO-SHIFT CLOCK LR-SHIFT-REG JK1 NOTJ1 JK2
    NOTJ2 JK3 NOTJ3 JK4 NOTJ4 MUX1 MUX2 MUX3 MUX4

.> (ar 7)          (* load the object just saved to a file and then deleted
                         from the application definition *)
Enter the path (include last /)./objs/:
Objects in the technology base:
"LOAD-SWITCH"
"TEST-STR-SUB"
"AND-1"

Enter the object name: load-switch
Enter the application name:  (DEMO):
Rule successfully applied.
application definition DEMO
```

```
  LR-BAR CLEAR LEFT-IN RIGHT-IN DI0 DI1 DI2 DI3 DO0 DO1 DO2
    DO3 DRIVE-SHIFT DO-SHIFT CLOCK LR-SHIFT-REG JK1 NOTJ1 JK2
    NOTJ2 JK3 NOTJ3 JK4 NOTJ4 MUX1 MUX2 MUX3 MUX4 LOAD-SWITCH

.> (ar 2) (* edit an object that includes sets and sequences *)
Enter the name of the object to edit:  (DEMO): drive-shift
"Enter CONTROLLEES"
"What do you want to do?"
1 ) ADD-ELEMENTS
2 ) DELETE-ELEMENTS
3 ) MAKE-SEQ-EMPTY
4 ) FINISH-EDITING
1
adding  elements
1 - LR-SHIFT-REG
2 - CLOCK
3 - LOAD-SWITCH
4 - LR-BAR
5 - CLEAR
6 - LEFT-IN
7 - RIGHT-IN
8 - DI0
9 - DI1
10 - DI2
11 - DI3
12 - DO3
13 - DO2
14 - DO1
15 - DO0
What index do you want to add:10
(a symbol): outputx                          (* adding an invalid element *)
"What do you want to do?"
1 ) ADD-ELEMENTS
2 ) DELETE-ELEMENTS
3 ) MAKE-SEQ-EMPTY
4 ) FINISH-EDITING
4
"Enter UPDATE"
"What do you want to do?"
1 ) ADD-ELEMENTS
2 ) DELETE-ELEMENTS
3 ) MAKE-SEQ-EMPTY
4 ) FINISH-EDITING
1
adding  elements
1 - setstate LOAD-SWITCH (POSITION, ON)
2 - setstate LR-BAR (POSITION, OFF)
3 - update CLEAR
4 - update CLOCK
5 - update LOAD-SWITCH
6 - update LR-BAR
```

```
7 - update LEFT-IN
8 - update RIGHT-IN
9 - update DI3
(* some output omitted *)
85 - update DO1
86 - update DO0
What index do you want to add:87

(* the following uses the domain model object hierarchy to determine what
   class of object is appropriate and, if it has subclasses, which subclass
   object should actually be built *)
"Enter which type of object you want to build"
1 ) CALL-OBJ
2 ) WHILE-STMT-OBJ
3 ) IF-STMT-OBJ
1
"Enter which type of object you want to build"
1 ) CONFIGURE-CALL-OBJ
2 ) STABILIZE-CALL-OBJ
3 ) INITIALIZE-CALL-OBJ
4 ) DESTROY-CALL-OBJ
5 ) SETSTATE-CALL-OBJ
6 ) SETFUNCTION-CALL-OBJ
7 ) CREATE-CALL-OBJ
8 ) UPDATE-CALL-OBJ
8
Enter OPERAND:  (*UNDEFINED*): outputx  (* adding a call to the
                                           invalid element *)
"What do you want to do?"
1 ) ADD-ELEMENTS
2 ) DELETE-ELEMENTS
3 ) MAKE-SEQ-EMPTY
4 ) FINISH-EDITING
4
"Enter INITIALIZE"
creating a seq of type NAME-VALUE-OBJ
Add another element? (y or n) n
Rule successfully applied.
application definition DEMO
  LR-BAR CLEAR LEFT-IN RIGHT-IN DI0 DI1 DI2 DI3 DO0 DO1 DO2
    DO3 DRIVE-SHIFT DO-SHIFT CLOCK LR-SHIFT-REG JK1 NOTJ1 JK2
    NOTJ2 JK3 NOTJ3 JK4 NOTJ4 MUX1 MUX2 MUX3 MUX4 LOAD-SWITCH

.> (pn 'drive-shift)
subsystem DRIVE-SHIFT is controls:
  LR-SHIFT-REG, CLOCK, LOAD-SWITCH, LR-BAR, CLEAR, LEFT-IN,
  RIGHT-IN, DI0, DI1, OUTPUTX, DI2, DI3, DO3, DO2,
  DO1, DO0
  imports: exports: initialize procedure: update procedure:
  setstate LOAD-SWITCH (POSITION, ON)
    setstate LR-BAR (POSITION, OFF) update CLEAR update CLOCK
```

```
    update LOAD-SWITCH update LR-BAR update LEFT-IN
    update RIGHT-IN update DI3 update DI2 update DI1 update DI0
    update LR-SHIFT-REG update DO3 update DO2 update DO1
    update DO0 setstate LOAD-SWITCH (POSITION, OFF)
    setstate LR-BAR (POSITION, OFF) update LOAD-SWITCH
    update LR-BAR update LR-SHIFT-REG update DO3 update DO2
    update DO1 update DO0 update LR-SHIFT-REG update DO3
    update DO2 update DO1 update DO0 update LR-SHIFT-REG
    update DO3 update DO2 update DO1 update DO0
    update LR-SHIFT-REG update DO3 update DO2 update DO1
    update DO0 setstate LOAD-SWITCH (POSITION, ON)
    setstate LR-BAR (POSITION, OFF) update CLEAR update CLOCK
    update LOAD-SWITCH update LR-BAR update LEFT-IN
    update RIGHT-IN update DI3 update DI2 update DI1 update DI0
    update LR-SHIFT-REG update DO3 update DO2 update DO1
    update DO0 setstate LOAD-SWITCH (POSITION, OFF)
    setstate LR-BAR (POSITION, ON) update LOAD-SWITCH
    update LR-BAR update DI3 update DI2 update DI1 update DI0
    update LR-SHIFT-REG update DO3 update DO2 update DO1
    update DO0 update LR-SHIFT-REG update DO3 update DO2
    update DO1 update DO0 update LR-SHIFT-REG update DO3
    update DO2 update DO1 update DO0 update LR-SHIFT-REG
    update DO3 update DO2 update DO1 update DO0 update OUTPUTX

(* now tests the semantics *)
.> (ar 15)

ERROR -- "Object OUTPUTX does not exist"   (* semantic error, outputx should
                                              exist in the object base but it
                                              does not. *)
Object: subsystem DRIVE-SHIFT

Rule successfully applied.
application definition DEMO
  LR-BAR CLEAR LEFT-IN RIGHT-IN DI0 DI1 DI2 DI3 DO0 DO1 DO2
    DO3 DRIVE-SHIFT DO-SHIFT CLOCK LR-SHIFT-REG JK1 NOTJ1 JK2
    NOTJ2 JK3 NOTJ3 JK4 NOTJ4 MUX1 MUX2 MUX3 MUX4 LOAD-SWITCH

(* the object outputx does not exist, so the application definition
   fails this check.  Now, remove it from the update algorithm *)

.> (ar 2)
Enter the name of the object to edit:  (DEMO): drive-shift
"Enter CONTROLLEES"
"What do you want to do?"
1 ) ADD-ELEMENTS
2 ) DELETE-ELEMENTS
3 ) MAKE-SEQ-EMPTY
4 ) FINISH-EDITING
2
Delete another element? (y or n) y
```

```
"Which one do you want to delete?"
1 ) LR-SHIFT-REG
2 ) CLOCK
3 ) LOAD-SWITCH
4 ) LR-BAR
5 ) CLEAR
6 ) LEFT-IN
7 ) RIGHT-IN
8 ) DI0
9 ) DI1
10 ) OUTPUTX
11 ) DI2
12 ) DI3
13 ) DO3
14 ) DO2
15 ) DO1
16 ) DO0
10                                (* remove outputx from controlees list *)
Delete another element? (y or n) n
"What do you want to do?"
1 ) ADD-ELEMENTS
2 ) DELETE-ELEMENTS
3 ) MAKE-SEQ-EMPTY
4 ) FINISH-EDITING
4
"Enter UPDATE"
"What do you want to do?"
1 ) ADD-ELEMENTS
2 ) DELETE-ELEMENTS
3 ) MAKE-SEQ-EMPTY
4 ) FINISH-EDITING
2
Delete another element? (y or n) y
"Which one do you want to delete?"
1 ) setstate LOAD-SWITCH (POSITION, ON)
2 ) setstate LR-BAR (POSITION, OFF)
3 ) update CLEAR
 (* some output omitted *)
85 ) update DO1
86 ) update DO0
87 ) update OUTPUTX
87                                   (* remove call to update outputx *)
Delete another element? (y or n) n
"What do you want to do?"
1 ) ADD-ELEMENTS
2 ) DELETE-ELEMENTS
3 ) MAKE-SEQ-EMPTY
4 ) FINISH-EDITING
4
"Enter INITIALIZE"
creating a seq of type NAME-VALUE-OBJ
```

```
Add another element? (y or n) n
Rule successfully applied.
application definition DEMO
  LR-BAR CLEAR LEFT-IN RIGHT-IN DI0 DI1 DI2 DI3 DO0 DO1 DO2
    DO3 DRIVE-SHIFT DO-SHIFT CLOCK LR-SHIFT-REG JK1 NOTJ1 JK2
    NOTJ2 JK3 NOTJ3 JK4 NOTJ4 MUX1 MUX2 MUX3 MUX4 LOAD-SWITCH

.> (ar 15)
 (* perform semantic checks and indicate sources for imports *)
 (* execute *)
```

D.2 Binary Array Multiplier

The following session demostrates the use of generic objects by building a binary array multiplier shown in Figure D.2.

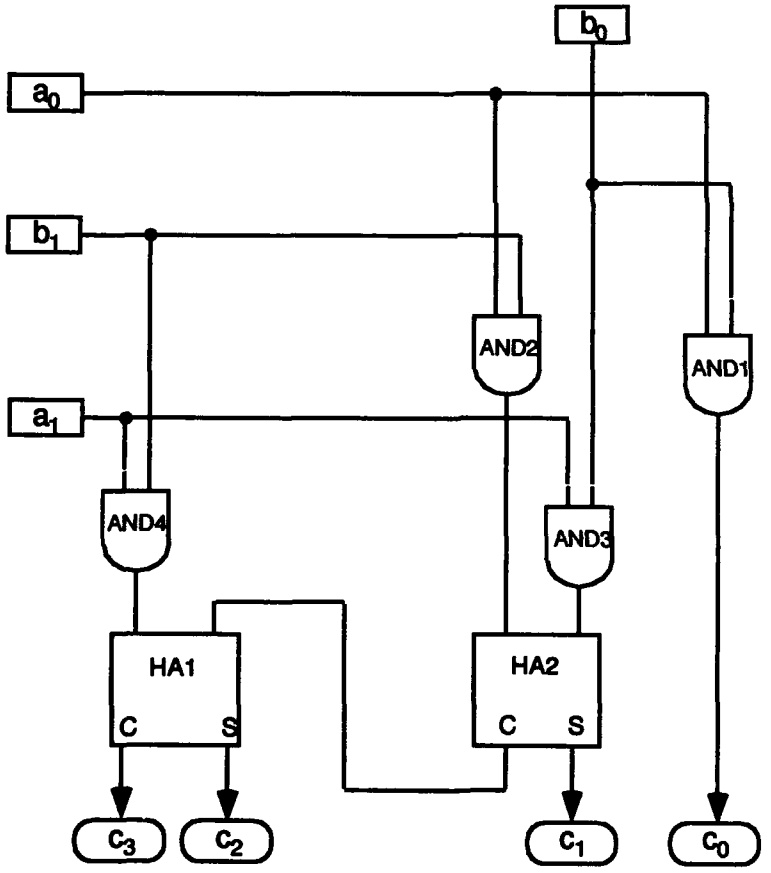

Figure D.2. Binary Array Multiplier

```
application definition bin-arr-mult1
% binary array multiplier using two generic half adders

switch a-zero position: on
switch a-one position: off

switch b-zero position: on
switch b-one position: off

led c-zero
led c-one
led c-two
led c-three

and-gate and-a1-b1  % and4 in figure
and-gate and-a1-b0  % and3 in figure
and-gate and-a0-b1  % and2 in figure
and-gate and-a0-b0  % and1 in figure

and-gate ha-1-and1
and-gate ha-1-and2
or-gate ha-1-or
not-gate ha-1-not

and-gate ha-2-and1
and-gate ha-2-and2
or-gate ha-2-or
not-gate ha-2-not is mil-spec

not-gate not-x-ha1
not-gate not-y-ha1
not-gate not-x-ha2
not-gate not-y-ha2

generic HA-1 is new /adders/gen-ha-no-int-3 ha-1-and1, ha-1-and2,
  ha-1-or, ha-1-not
generic HA-2 is new /adders/gen-ha-no-int-3 ha-2-and1, ha-2-and2,
  ha-2-or, ha-2-not


subsystem bin-ar-mult is
  controls: HA-1, HA-2,
            and-a1-b1, and-a1-b0, and-a0-b1, and-a0-b0,
            a-zero, a-one, b-zero, b-one,
            c-zero, c-one, c-two, c-three,
            not-x-ha1, not-y-ha1, not-x-ha2, not-y-ha2

  update procedure:
  update a-zero
  update a-one
  update b-zero
```

```
  update b-one
  update and-a0-b1
  update and-a0-b0
  update and-a1-b1
  update and-a1-b0

  update not-x-ha1
  update not-y-ha1
  update not-x-ha2
  update not-y-ha2


  update HA-1
  update HA-2

  update c-zero
  update c-one
  update c-two
  update c-three

application run-bam is
 controls: bin-ar-mult
 update procedure:
   update bin-ar-mult



.> (ar 9)                                    (* parse in the input file *)
Enter the full name of the file: ./DSL/tests/bin-array-mult
Rule successfully applied.
application definition DEMO
  LR-BAR CLEAR LEFT-IN RIGHT-IN DI0 DI1 DI2 DI3 DO0 DO1 DO2
    DO3 DRIVE-SHIFT DO-SHIFT CLOCK LR-SHIFT-REG JK1 NOTJ1 JK2
    NOTJ2 JK3 NOTJ3 JK4 NOTJ4 MUX1 MUX2 MUX3 MUX4 LOAD-SWITCH

.> (mcn 'bin-arr-mult1)
application definition BIN-ARR-MULT1
  A-ZERO A-ONE B-ZERO B-ONE C-ZERO C-ONE C-TWO C-THREE
    AND-A1-B1 AND-A1-B0 AND-A0-B1 AND-A0-B0 HA-1-AND1 HA-1-AND2
    HA-1-OR HA-1-NOT HA-2-AND1 HA-2-AND2 HA-2-OR HA-2-NOT
    NOT-X-HA1 NOT-Y-HA1 NOT-X-HA2 NOT-Y-HA2 HA-1 HA-2
    BIN-AR-MULT RUN-BAM

.> (pup 'ha-1)              (* HA-1 is a generic instance, it has not been
                               instantiated yet into a subsystem *)
HA-1 - the generic-inst
   re::class:  generic-inst
   parent-expr: #2<BIN-ARR-MULT1 - the spec-obj>
   name:  ha-1
   generic-parameters: [HA-1-AND1, HA-1-AND2, ..]
   generic-to-be-used:  /adders/gen-ha-no-int-3
```

```
.> (ar 13)                    (* complete application definition *)
Enter the application name:  (BIN-ARR-MULT1):
Instantiating Generic generic HA-1 is new /ADDERS/GEN-HA-NO-INT-3
  HA-1-AND1, HA-1-AND2, HA-1-OR, HA-1-NOT
Instantiating Generic generic HA-2 is new /ADDERS/GEN-HA-NO-INT-3
  HA-2-AND1, HA-2-AND2, HA-2-OR, HA-2-NOT
Rule successfully applied.
application definition BIN-ARR-MULT1
  A-ZERO A-ONE B-ZERO B-ONE C-ZERO C-ONE C-TWO C-THREE
    AND-A1-B1 AND-A1-B0 AND-A0-B1 AND-A0-B0 HA-1-AND1 HA-1-AND2
    HA-1-OR HA-1-NOT HA-2-AND1 HA-2-AND2 HA-2-OR HA-2-NOT
    NOT-X-HA1 NOT-Y-HA1 NOT-X-HA2 NOT-Y-HA2 BIN-AR-MULT RUN-BAM
    HA-1 HA-2

.> (pn 'ha-1)  (* now HA-1 is a subsystem, notice that the import
                  sources are defined even though the semantic checks
                  have not been executed.  This information was saved
                  with the generic object. *)

subsystem HA-1 is controls:
  HA-1-AND1, HA-1-AND2, HA-1-OR, HA-1-NOT
  imports:
    IN1 SIGNAL BOOLEAN HA-1-AND1 ( OUT1 DRIVE-ADD2 NEGATE-X)
      IN2 SIGNAL BOOLEAN HA-1-AND1 ( OUT1 DRIVE-ADD2 NEGATE-Y)
      IN1 SIGNAL BOOLEAN HA-1-AND2
        ( OUT1 DRIVE-ADD2 SWITCH-XINPUT)
      IN2 SIGNAL BOOLEAN HA-1-AND2
        ( OUT1 DRIVE-ADD2 SWITCH-YINPUT)
      IN1 SIGNAL BOOLEAN HA-1-OR ( OUT1 HA-1 HA-1-AND1)
      IN2 SIGNAL BOOLEAN HA-1-OR ( OUT1 HA-1 HA-1-AND2)
      IN1 SIGNAL BOOLEAN HA-1-NOT ( OUT1 HA-1 HA-1-OR)
  exports:
    OUT1 SIGNAL BOOLEAN NIL HA-1-AND1
      OUT1 SIGNAL BOOLEAN T HA-1-AND2
      OUT1 SIGNAL BOOLEAN T HA-1-OR
      OUT1 SIGNAL BOOLEAN NIL HA-1-NOT
  initialize procedure: update procedure:
  update HA-1-AND1 update HA-1-AND2 update HA-1-OR
    update HA-1-NOT

(* the following shows how the same generic instance could have been
   built interactively instead of through the grammar *)

.> (ar 10)
Enter the path name (include last /)./generics/: adders/
Here are the existing generics:
"GEN-4-BIT-ADDER"
"GEN-FULL-ADDER"
"GEN-HA-NO-INT-3"
"GEN-HA-NO-INT"
```

```
"GENERIC-HA1"
"GEN-HA-PLAIN"
Enter the name of the generic to instantiate: gen-ha-no-int-3
Enter the application name:  (BIN-ARR-MULT1):
Enter the name for the generic instance: test-half-adder
Enter a symbol for GEN-HA1-AND1:  (*UNDEFINED*): ha-1-and1
Enter a symbol for GEN-HA1-AND2:  (*UNDEFINED*): ha-1-and2
Enter a symbol for GEN-HA1-OR1:  (*UNDEFINED*): ha-1-or4
Enter a symbol for GEN-HA1-NOT1:  (*UNDEFINED*): ha-1-not
Rule successfully applied.
application definition BIN-ARR-MULT1
  A-ZERO A-ONE B-ZERO B-ONE C-ZERO C-ONE C-TWO C-THREE
    AND-A1-B1 AND-A1-B0 AND-A0-B1 AND-A0-B0 HA-1-AND1 HA-1-AND2
    HA-1-OR HA-1-NOT HA-2-AND1 HA-2-AND2 HA-2-OR HA-2-NOT
    NOT-X-HA1 NOT-Y-HA1 NOT-X-HA2 NOT-Y-HA2 BIN-AR-MULT RUN-BAM
    HA-1 HA-2 TEST-HALF-ADDER

.> (ar 13)                    (* the new instance must be instantiated *)
Enter the application name:  (BIN-ARR-MULT1):
HA-1-OR4 should be an object, but it is not in the object base
Invalid generic object generic TEST-HALF-ADDER is new ADDERS/GEN-HA-NO-INT-3
  HA-1-AND1, HA-1-AND2, HA-1-OR4, HA-1-NOT
Do you want to edit the object?(y or n) y
(* ha-1-or4 was not in the object base *)
HA-1-OR4 should be an object, but it is not in the object base
Enter a symbol for GEN-HA1-AND1:  (HA-1-AND1):
Enter a symbol for GEN-HA1-AND2:  (HA-1-AND2):
Enter a symbol for GEN-HA1-OR1:  (HA-1-OR4): ha-1-or
Enter a symbol for GEN-HA1-NOT1:  (HA-1-NOT):
Enter the name:  (TEST-HALF-ADDER):
Rule successfully applied.

.> (ar 4)
What object do you want to delete?:  (BIN-ARR-MULT1): test-half-adder

Are you sure you want to detete TEST-HALF-ADDER (y or n) y
Rule successfully applied.
```

Appendix E. *Code*

```
!! in-package("RU")
!! in-grammar('user)

#||
 File name: dm-ocu.re

 Description: Domain Model for the OCU model
||#


% OBJECT CLASSES:

var World-Obj                : object-class subtype-of user-Object

  var Spec-Obj               : object-class subtype-of World-Obj
    % A high-level object that ties together all of the parts of an
    % application definition

  var Spec-Part-Obj          : object-class subtype-of World-Obj
    % Spec-Parts describe all of the senetences used by the application
    % specialist to build an application definition

    var Incomplete-Obj       : object-class subtype-of Spec-Part-Obj
    var Generic-Inst         : object-class subtype-of Spec-Part-Obj
    var Load-Obj             : object-class subtype-of Spec-Part-Obj

    var Component-Obj        : object-class subtype-of Spec-Part-Obj
      % Component-Obj's are all the parts of a final definition

      var Application-Obj    : object-class subtype-of Component-Obj
      var Subsystem-Obj      : object-class subtype-of Component-Obj
      var Primitive-Obj      : object-class subtype-of Component-Obj


  var Statement-Obj          : object-class subtype-of World-Obj
    var If-Stmt-Obj          : object-class subtype-of Statement-Obj
    var While-Stmt-Obj       : object-class subtype-of Statement-Obj
    var Call-Obj             : object-class subtype-of Statement-Obj
      var Update-Call-Obj       : object-class subtype-of Call-Obj
      var Create-Call-Obj       : object-class subtype-of Call-Obj
      var SetFunction-Call-Obj  : object-class subtype-of Call-Obj
      var SetState-Call-Obj     : object-class subtype-of Call-Obj
      var Destroy-Call-Obj      : object-class subtype-of Call-Obj
      var Initialize-Call-Obj   : object-class subtype-of call-obj
      var Stabilize-Call-Obj    : object-class subtype-of call-obj
      var Configure-Call-Obj    : object-class subtype-of call-obj
```

```
  var Import-Export-Obj      : object-class subtype-of World-Obj
  var Import-Obj             : object-class subtype-of Import-Export-Obj
  var Export-Obj             : object-class subtype-of Import-Export-Obj
  var Name-Value-Obj         : object-class subtype-of World-Obj
  var Source-Obj             : object-class subtype-of World-Obj

  var Generic-Obj            : object-class subtype-of World-Obj


%%% ATTRIBUTES:

% Spec-Obj:
var Spec-Parts   : map(Spec-Obj, seq(Spec-Part-Obj))
    computed-using
          Spec-Parts(x) = []


% Incomplete-Obj:
var Obj-Type     : map(Incomplete-Obj, symbol)     = {| |}


% Generic-Inst:
var Generic-To-Be-Used : map(Generic-Inst, symbol) = {| |}
var Generic-Parameters : map(Generic-Inst, seq(any-type))
  computed-using
    Generic-Parameters(x) = []


% Load-Obj:
var Object-To-Load     : map(Load-Obj, symbol)     = {| |}


% Application-Obj:
 var Application-Components   : map(Application-Obj, seq(symbol))
   computed-using
     Application-Components(x) = []

 var Application-update       : map(Application-Obj, seq(Statement-Obj))
   computed-using
     Application-update(x) = []


% Subsystem-Obj:
var Controllees              : map(Subsystem-Obj, seq(symbol))
   computed-using
     controllees(x) = []

%%    changed seq to set in import-Area and Export-Area ...
var Import-Area              : map(Subsystem-Obj, set(Import-Obj))
   computed-using
    Import-Area(x) = {}
```

```
var Export-Area                : map(Subsystem-Obj, set(Export-Obj))
   computed-using
     Export-Area(x) = {}

var Update                     : map(Subsystem-Obj, seq(Statement-Obj))
   computed-using
     Update(x) = []
var Initialize                 : map(subsystem-obj, seq(name-value-obj))
   computed-using
     Initialize(x) = []


% Statements:

 % If-Stmt-Obj

 var If-Cond                  : map(If-stmt-Obj, Expression)       = {| |}
 var Then-Stmts               : map(If-stmt-Obj, seq(Statement-Obj))
  computed-using
     Then-Stmts(x) = []
 var Else-Stmts               : map(If-stmt-Obj, seq(Statement-Obj))
  computed-using
     Else-Stmts(x) = []


 % While-Stmt-Obj:

 var  While-cond               : map(While-stmt-Obj, expression)    = {| |}
 var  While-stmts              : map(While-stmt-Obj, seq(Statement-Obj))
   computed-using
      While-stmts(x) = []


 % Call-Obj:
 var operand                  : map(Call-Obj, symbol) = {| |}

  % Create-Call-Obj:
  var object-type             : map(create-Call-Obj, symbol) = {||}

  % SetFunction-Call-Obj:
  var function-name           : map(SetFunction-Call-Obj, symbol) = {||}
  var coefficients            : map(SetFunction-Call-Obj, set(name-value-Obj))
    computed-using
      coefficients(x) = {}


  % SetState-Call-Obj:
  var state-changes           : map(SetState-Call-Obj, set(name-value-Obj))
    computed-using
       state-changes(x) = {}
```

```
% Import-Obj:
var import-name              : map(Import-Obj, symbol)          = {||}
var import-category          : map(Import-Obj, symbol)          = {||}
var import-type-data         : map(Import-Obj, symbol)          = {||}
var consumer                 : map(Import-Obj, symbol)          = {||}
var Source                   : map(Import-Obj, set(Source-Obj))
  computed-using
    Source(x) = {}


% Export-Obj:
var export-name              : map(Export-Obj, symbol)   = {||}
var export-category          : map(Export-Obj, symbol)   = {||}
var export-type-data         : map(Export-Obj, symbol)   = {||}
var value                    : map(Export-Obj, any-type) = {||}
var producer                 : map(Export-Obj, symbol)   = {||}


% Name-Value-Obj:
var Name-value-Name          : map(Name-value-Obj, symbol) = {||}
var Name-value-value         : map(Name-value-Obj, any-type) = {||}


% Source-Obj:
var Source-Subsystem        : map(Source-Obj, symbol)     = {||}
var Source-Object           : map(Source-Obj, symbol)     = {||}
var Source-Name             : map(Source-Obj, symbol)     = {||}

% Generic-Obj:
var Obj-Instance         : map(Generic-Obj, Symbol)   = {| |}
var Placeholder-IDs      : map(Generic-Obj, seq(any-type))
  computed-using
    Placeholder-IDs(x) = []

var Placeholder-Type     : map(Generic-Obj, seq(symbol))
  computed-using
    Placeholder-Type(x) = []

var Obj-List             : map(Generic-Obj, seq(symbol))
  computed-using
    Obj-List(x) = []



%-------------------------------------------------------------------


% Code for Boolean-expressions
 var Expression : object-class subtype-of World-Obj
```

```
var Literal-Expression : object-class subtype-of expression

var Identifier         : object-class subtype-of Literal-Expression
var Boolean-Literal    : object-class subtype-of Literal-Expression
   var True-Literal    : object-class subtype-of Boolean-Literal
   var False-Literal   : object-class subtype-of Boolean-Literal
var Number-Literal     : object-class subtype-of Literal-Expression
  var Integer-Literal  : object-class subtype-of Number-Literal
  var Real-Literal     : object-class subtype-of Number-Literal
var String-Literal     : object-class subtype-of Literal-Expression


var Unary-Expression : object-class subtype-of Expression
   var Not-Exp       : object-class subtype-of Unary-Expression
   var abs-exp       : object-class subtype-of unary-expression
   var negate-exp    : object-class subtype-of unary-expression
   var positive-exp  : object-class subtype-of unary-expression

var Binary-Expression   : object-class subtype-of Expression
   var Or-Exp           : object-class subtype-of Binary-Expression
   var And-Exp          : object-class subtype-of Binary-Expression
   var Equal-Exp        : object-class subtype-of Binary-Expression
   var Not-Equal-Exp    : object-class subtype-of Binary-Expression
   var LT-Exp           : object-class subtype-of Binary-Expression
   var LTE-Exp          : object-class subtype-of Binary-Expression
   var GT-Exp           : object-class subtype-of Binary-Expression
   var GTE-Exp          : object-class subtype-of Binary-Expression

   var add-exp          : object-class subtype-of Binary-Expression
   var subtract-exp     : object-class subtype-of Binary-Expression
   var multiply-exp     : object-class subtype-of Binary-Expression
   var divide-exp       : object-class subtype-of Binary-Expression
   var mod-exp          : object-class subtype-of Binary-Expression
   var exponential-exp  : object-class subtype-of Binary-Expression


%%  Attributes for expressions:
var Id-Name            : map(Identifier, symbol) = {| |}
var Id-Source          : map(Identifier, import-export-obj) = {| |}

var Int-value          : map(Integer-Literal, integer) = {| |}
var Real-value         : map(Real-Literal, real) = {| |}
var Boolean-value      : map(Boolean-Literal, boolean) = {| |}
var String-value       : map(String-Literal, string) = {| |}

var Argument1 : map(Binary-Expression, Expression) = {| |}
var Argument2 : map(Binary-Expression, Expression) = {| |}
var Argument  : map(Unary-Expression,  Expression) = {| |}

%% Tree Attributes For Expressions
```

```
Form Expression-Attrs
  define-tree-attributes('Binary-Expression, {'Argument1, 'Argument2});
  define-tree-attributes('Unary-Expression, {'Argument})


Form Define-AST
  define-tree-attributes('While-Stmt-Obj, {'While-Cond, 'While-Stmts});
  define-tree-attributes('If-Stmt-Obj,
                         {'If-Cond, 'Then-Stmts, 'Else-Stmts});
  define-tree-attributes('Call-Obj, {'Operand});
  define-tree-attributes('SetFunction-Call-Obj,
                         {'Function-Name, 'Coefficients});
  define-tree-attributes('SetState-Call-Obj, {'state-changes});
  define-tree-attributes('Application-Obj,
                         {'Application-Components, 'Application-Update});
  define-tree-attributes('Subsystem-Obj,
                         {'Controllees, 'Update, 'Initialize,
                          'Export-Area, 'Import-Area});
  define-tree-attributes('Spec-Obj, {'Spec-Parts});
  define-tree-attributes('Import-Obj,
                         {'Import-Name, 'Import-Category,
                         'Import-Type-Data, 'Source, 'Consumer});
  define-tree-attributes('Export-Obj,
                         {'Export-Name, 'Export-Category,
                         'Export-Type-Data, 'Value, 'Producer});
  define-tree-attributes('Generic-Obj,
                         {'Obj-Instance, 'Placeholder-Ids, 'Obj-List})


 form Make-Names-Unique
  unique-names-class('Spec-Obj, true);
  unique-names-class('Application-Obj, true);
  unique-names-class('Subsystem-Obj, true);
  unique-names-class('Generic-Obj, true);
  unique-names-class('Generic-Inst, true);
  unique-names-class('Load-Obj, true);
  unique-names-class('Incomplete-Obj, true)
```

```
!! in-package("RU")
!! in-grammar('syntax)

#||
 File name:  gram-ocu.re

 Description: The OCU grammar - all of the domain-independent productions,
  most of which describe the OCU model
  NOTE: If you change this file, you must also recompile the
        domain-specific grammar.  If no changes are made to that
        grammar, erase its .fasl4 file to make sure it recompiles.
        Otherwise, you won't see the changes made to the OCU grammar.

||#


grammar OCU

  no-patterns

  start-classes Spec-Obj, Subsystem-obj, Application-Obj,
                Incomplete-obj, Load-Obj, Generic-Obj

  file-classes Spec-Obj, Subsystem-obj, Application-Obj,
               Incomplete-obj, Load-Obj, Generic-Obj

  productions

    Spec-Obj        ::= ["application definition" name {Spec-Parts + ""} ]
                                        builds Spec-Obj,


    Application-Obj      ::= ["application" name "is"
                              "controls:" application-components * ","
                               "update procedure:"
                               application-update * ""]
                                        builds Application-Obj,


    Subsystem-Obj     ::= ["subsystem" name "is"
                           "controls:" Controllees * ","
                           {["imports:" Import-Area * ""]}
                           {["exports:" Export-Area * ""]}
                           {["initialize procedure:"
                                initialize * ""]}
                           "update procedure:"
                            update * "" ]      builds Subsystem-Obj,

    Import-Obj       ::= [import-name import-category  import-type-data
                          consumer  "(" [source * ""] ")"]
                                        builds Import-Obj,
```

```
Export-Obj        ::= [export-name export-category  export-type-data
                       value  producer]    builds Export-Obj,

Source-Obj        ::= [Source-Name Source-Subsystem Source-Object]
                                           builds Source-Obj,

Generic-Inst      ::= ["generic" name "is" "new" Generic-To-Be-Used
                       Generic-Parameters * ","]
                                           builds Generic-Inst,

Incomplete-Obj    ::= ["object" obj-type "," name]
                                           builds Incomplete-obj,

Load-Obj          ::= ["load" Object-To-Load]
                                           builds Load-Obj,

If-Stmt-Obj       ::= ["if" if-cond "then" Then-Stmts + ""
                      {["else" Else-Stmts + ""]}
                      "end" "if" ]         builds If-Stmt-Obj,

While-Stmt-Obj  ::= ["while" while-cond "loop"
                       While-Stmts * ""
                       "end"  "while" ]    builds While-Stmt-Obj,


Update-Call-Obj       ::= ["update" operand] builds Update-Call-Obj,


Create-Call-Obj       ::= ["create" operand object-type]
                                           builds Create-Call-Obj,


SetFunction-Call-Obj ::= ["setfunction" operand function-name
                              Coefficients * ""]
                                           builds SetFunction-Call-Obj,


SetState-Call-Obj     ::= ["setstate" operand
                              State-Changes * ""]
                                           builds SetState-Call-Obj,


Destroy-Call-Obj      ::= ["destroy" operand]
                                           builds Destroy-Call-Obj,

Initialize-Call-Obj   ::= ["initialize" operand]
                                           builds Initialize-Call-Obj,

Configure-Call-Obj    ::= ["configure" operand]
                                           builds Configure-Call-Obj,
```

```
  Stabilize-Call-Obj   ::= ["stabilize" operand]
                                        builds Stabilize-Call-Obj,

  Name-Value-Obj       ::= ["(" name-value-name ","
                               name-value-value ")"]
                                        builds Name-Value-Obj,


  Generic-Obj      ::= ["generic-obj" name Obj-Instance
                       "ids:"  {Placeholder-IDs + ","}
                       "types:" {Placeholder-Type + ","}
                       "objects:" {Obj-List + ","}    ]
                                       builds Generic-Obj,


And-Exp ::= [argument1 "and" argument2]      builds And-Exp,
Or-Exp  ::= [argument1 "or"  argument2]      builds or-Exp,

Equal-Exp     ::= [argument1 "="  argument2] builds Equal-Exp,
Not-Equal-Exp ::= [argument1 "/=" argument2] builds Not-Equal-Exp,
LT-Exp        ::= [argument1 "<"  argument2] builds LT-Exp,
LTE-Exp       ::= [argument1 "<=" argument2] builds LTE-Exp,
GT-Exp        ::= [argument1 ">"  argument2] builds GT-Exp,
GTE-Exp       ::= [argument1 ">=" argument2] builds GTE-Exp,

Add-Exp       ::= [argument1 "+" argument2]  builds Add-Exp,
Subtract-Exp  ::= [argument1 "-" argument2]  builds Subtract-Exp,


Multiply-Exp    ::= [argument1 "*"   argument2] builds Multiply-Exp,
Divide-Exp      ::= [argument1 "/"   argument2] builds Divide-Exp,
Mod-Exp         ::= [argument1 "mod" argument2] builds Mod-Exp,
Exponential-Exp ::= [argument1 "**"  argument2] builds Exponential-Exp,

Abs-Exp      ::= ["abs" argument] builds Abs-Exp,
Not-Exp      ::= ["not" argument] builds Not-Exp,
Negate-Exp   ::= ["-" argument]   builds Negate-Exp,
Positive-Exp ::= ["+" argument]   builds Positive-Exp,


Identifier      ::= [Id-Name]      builds Identifier,
Integer-Literal ::= [Int-Value]    builds Integer-Literal,
Real-Literal    ::= [Real-Value]   builds Real-Literal,
String-Literal  ::= [String-Value] builds String-Literal,
True-Literal    ::= ["true"]       builds True-Literal,
False-Literal   ::= ["false"]      builds False-Literal


symbol-start-chars
   "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

```
 symbol-continue-chars
    "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ-0123456789"

precedence

   for expression brackets "(" matching ")"
     (same-level "and", "or" associativity left),
     (same-level "<", "<=", "=", ">=", ">", "/=" associativity none),
     (same-level "+","-"  associativity left),
     (same-level "*", "/", "mod"  associativity left),
     (same-level "not"  associativity none),
     (same-level "abs"  associativity none),
     (same-level "**" associativity right)

 end
```

```
!! in-package("RU")
!! in-grammar('user)

#||
 File name: modify-obj.re

 Description:
  The functions in this file allow the user to modify objects - delete
  objects, edit existing objects, and add new objects

 Rules:
   Edit-An-Object
   Add-An-Object
   Delete-An-Object

 Functions:
   Modify-Some-Object
   Modify-Object
   Get-Attr-Name
   Update-Attr
   Find-Subnode
   Make-New-Object
   Read-Set
   Edit-Set
   Read-Seq
   Edit-Seq
   Is-Valid-New-Type
   Add-Object
   Delete-Object

||#

var Attrs-To-Ignore : set(symbol) =
             {'import-area, 'export-area, 'coefficients}
% attributes that shouldn't be edited

%% Rules that can be performed by the user
%-------------------------------------------------------------------------
rule Edit-An-Object(X: object)   % Do not change the name to edit-object,
                                 % it exists already)
  true --> Modify-Some-Object(x)

rule Add-An-Object(X: Object)
  true --> Add-Object(x)

rule Delete-An-Object(X: Object)
  true --> Delete-Object(X)

%-------------------------------------------------------------------------
" Asks the user for the name of the object, checks that it is a subclass
 of component object (i.e., its a subsystem or primitive domain object),
```

```
 and then modifies the object"

function Modify-Some-Object(X : object) =

  let (obj-to-edit : symbol =
         Read-Symbol-Default("Enter the name of the object to edit",
                             Name(x)))

    let (Edit-Obj : object = find-object('component-obj, obj-to-edit))
      if defined?(Edit-Obj) then
         Changes-Made <- true;
         Edit-Obj <- Modify-Object (Edit-Obj)
      else
        format(t, "Object ~s is not a current object
                   that can be edited~%", obj-to-edit)



%-------------------------------------------------------------------------
" Given an object, goes through each attribute that's not one of the
 predefined attributes and gets a value for it"

function Modify-Object (obj : object) : object =
  (enumerate atr over return-attribute-list(obj) do
   if name(atr) ~in Attrs-To-Ignore and
          String-To-Symbol(Get-Attr-Name(Name(Atr)), "RU") ~in
             Attrs-To-Ignore then
    update-attr(obj, atr)
  );
  Obj

%-------------------------------------------------------------------------
"If the given name has the object class name prepended, return just the
 attribute name"

function Get-Attr-Name(Full-Name : symbol) : string =

 let (atr-string = symbol-to-string(full-name))
   let(just-name : string =
          arb( {atr-name | (atr-name, first-part)
            atr-string = [$first-part, $atr-name] &
            Is-Valid-New-Type(string-to-symbol(first-part, "RU")) }) )
     if undefined?(Just-Name) then
       symbol-to-string(full-name)
     else
       if first(just-name) = #\- then
            Rest(just-Name)
       else
           Just-name
```

```
%-----------------------------------------------------------------------
" Given an object and an attribute, finds the data type of the attribute,
  calls the appropriate read function, and stores the value in the
  attribute.  If the attribute is an object, it first creates an object
  of that type
  (re::bindingname(re::range-type(re::data-type(attribute-binding))))
  and then gets the information for that object "

function Update-Attr(for-obj : object, attrib : re::binding) =

   let (attr-type : symbol = tell-type( attrib),
          prompt : string = concat("Enter ",
                                   Get-Attr-Name(name(attrib))),
    current-value : any-type = Retrieve-Attribute(for-obj, attrib) )

     if attr-type = 'real then
         store-attribute(for-obj, attrib,
                         read-real-default(prompt, current-value))
     elseif attr-type = 'integer then
         store-attribute(for-obj, attrib,
                        read-integer-default(prompt, current-value))
     elseif  attr-type = 'string then
         store-attribute(for-obj, attrib,
                        read-string-default(prompt, current-value))
     elseif  attr-type = 'boolean then
         store-attribute(for-obj, attrib,
                        read-boolean-default(prompt, current-value))
     elseif  attr-type = 'symbol then
         store-attribute(for-obj, attrib,
                        read-symbol-default(prompt, current-value))
     elseif attr-type = 'any-type then
         store-attribute(for-obj, attrib,
                        read-any-type-default(prompt, current-value))
     elseif attr-type = 'object then
         if defined?(current-value) then
           % object already exists, just update it
           store-attribute(for-obj, attrib, Modify-Object(current-value))
         else
           store-attribute(for-obj, attrib,
                Make-New-Object(
                    re::ref-to(re::range-type(re::data-type(attrib)))))
     elseif  attr-type = 'seq then
         format(t, "~s~%", prompt);  % Read-Seq doesn't print this prompt
         store-attribute(for-obj, attrib, read-seq(attrib, current-value))
     elseif attr-type = 'set then
         format(t, "~s~%", prompt); % Read-Set doesn't print this prompt
         store-attribute(for-obj, attrib, read-set(attrib, current-value))
      else
         format(t, "Unrecognized type ~s ~%", attr-type)
```

```
%-------------------------------------------------------------------------
"Sets the spec object to be the parent of the new object
 could instead have a rule:
  true --> Parent-Expr(Kid) = parent & kid in spec-parts(parent)"

function  Set-To-Parent(Kid, Parent : object) =
  Spec-Parts(Parent) <- append(Spec-Parts(Parent), Kid)


%-------------------------------------------------------------------------
" Removes the kid from the parent object"

function  Remove-From-Parent(Kid, Parent : object) =
  Spec-Parts(Parent) <-
      [objs | (objs : object) objs in Spec-Parts(Parent) &
                 name(objs) ~= Name(Kid)]  %Remove from application


%-------------------------------------------------------------------------
" Finds all of the subclasses of of an attribute and if more than one
  exists, asks the user which one he wants.  Class-Subclasses returns
  the current class"

function Find-SubNode (attrib : re::binding) : re::binding =

 % first, get the right object class (it may have subclasses)
  let (subnodes : seq(re::binding) =
          set-to-seq(Class-Subclasses(attrib, false) less attrib))
       % remove the current class (attrib) from the list of all subclasses
    let (Object-wanted : re::binding = re::*undefined*)
         % the type of obj to create

      (if SubNodes = nil then
         % if it doesn't have any subtypes, the set is nil
         Object-wanted <- attrib

       elseif size(subnodes) > 1 then
          % it has subobject, find out which one to use

         let (response : integer =
             Make-Object-Menu(subnodes,
                    "Enter which type of object you want to build"))

           Object-wanted <- subnodes(response)

       else
         % there's only one subtype of object,
         % this probably shouldn't happen
         Object-wanted <- subnodes(1)
      );
      (let (subsubnodes : set(re::binding) =
```

```
            Class-Subclasses(object-wanted, false))
        if subsubnodes ~= nil and-then size(subsubnodes) > 1 then
            % The object selected has subnodes, find the
            % object class at this level
          object-wanted <- find-subnode(object-wanted)
      );
      object-wanted

%-------------------------------------------------------------------------
" Given an attribute that represents an object, creates an object of that
  type and gets all of the attribute data "

function Make-New-Object( attrib : re::binding) : object =

   let (temp-obj : object = make-object(name(Find-Subnode(attrib))))
     Temp-Obj <- Modify-Object(temp-obj);
     Temp-obj

%-------------------------------------------------------------------------
" Given a set, edits it"

function Edit-Set(attr : re::binding,
           current-set : set(any-type)) :  set(any-type) =
  let (Choice-List : seq(symbol) = ['Add-Elements, 'Delete-Elements,
                                    'Make-Set-Empty, 'Finish-Editing])
    let (Choice : integer       = 0,
       Temp-Set : set(any-type) = Current-Set)

      (while Choice ~= 4 do
        Choice <- Make-Menu(Choice-List, "What do you want to do?");
        if Choice = 1 then
           (format(t, "adding  elements~%");
           let  (of-type : symbol = Tell-Set-Seq-Type(attr))

             (while Read-Yes-Or-No( "Add another element? " ) do
                if of-type = 'integer then
                  temp-set <- temp-set with Read-Integer("(an integer)")
                elseif of-type = 'real then
                  temp-set <- temp-set with Read-Real("(a real number)")
                elseif of-type = 'string then
                  temp-set <- temp-set with Read-String("(a string)")
                elseif of-type = 'symbol then
                  temp-set <- temp-set with Read-Symbol("(a symbol)")
                elseif of-type = 'boolean then
                  temp-set <- temp-set with Read-Boolean("(a boolean)")
                elseif of-type = 'any-type then
                  temp-set <- temp-set with Read-Any-Type("(any-type)")
                else % must be an obj
                  temp-set <- temp-set with
                            Make-New-Object(Tell-Set-Seq-Binding(attr))
```

```
          ) % end while
        ) % end if choice = 1

      elseif Choice = 2 then
         if empty(Temp-Set) then
           format(t, "No elements to delete~%")
         else
           format(t, "deleting elements~%");
           let (Set-Elements : seq(any-type) = set-to-seq(Temp-Set))
            (while ~empty(Temp-Set) and-then
                      Read-Yes-Or-No( "Delete another element? " ) do

              if Tell-Set-Seq-Type(attr) in
                 {'set, 'seq, 'symbol, 'real, 'integer, 'boolean} then
                 let (One-To-Delete : integer = Make-Menu(Set-Elements,
                                    "Which one do you want to delete?"))
                    true --> Set-Elements(One-To-Delete) ~in Temp-Set;
                    Set-Elements <- set-to-seq(Temp-Set)
              else
                 let (One-To-Delete : integer =
                                Make-Object-Menu(Set-Elements,
                                 "Which one do you want to delete?"))
                    true --> Set-Elements(One-To-Delete) ~in Temp-Set;
                    Set-Elements <- set-to-seq(Temp-Set)
            ) % end while

      elseif Choice = 3 then
         format(t, "making set empty~%");
          temp-set <- {}

   );
   temp-set

%-------------------------------------------------------------------------
" Reads in a group of items of the given type and puts them into a set.
  Since a set may already exist, it first asks if the user wants to change
  the original set"

function Read-Set(attr : re::binding,
          current-set : set(any-type)) : set(any-type) =

  if ~empty(Current-Set) then
     Edit-Set(attr, current-set)
    else
      let (temp-set : set(any-type) = {},
           of-type : symbol = Tell-Set-Seq-Type(attr))

        format(t, "creating a set of type ~s ~%", of-type);
        (while Read-Yes-Or-No( "Add another element? " ) do
          if of-type = 'integer then
             temp-set <- temp-set with Read-Integer("(an integer)")
```

```
            elseif of-type = 'real then
              temp-set <- temp-set with Read-Real("(a real number)")
            elseif of-type = 'string then
              temp-set <- temp-set with Read-String("(a string)")
            elseif of-type = 'symbol then
              temp-set <- temp-set with Read-Symbol("(a symbol)")
            elseif of-type = 'boolean then
              temp-set <- temp-set with Read-Boolean("(a boolean)")
            elseif of-type = 'any-type then
              temp-set <- temp-set with Read-Any-Type("(any-type)")
            else % must be an obj
              temp-set <- temp-set with
                        Make-New-Object(Tell-Set-Seq-Binding(attr))

          ); % end while
          temp-set


%-------------------------------------------------------------------------
" Given a seq, edits it"

function Edit-Seq(attr : re::binding,
            current-seq : seq(any-type)) : seq(any-type) =
  let (Choice-List : seq(symbol) = ['Add-Elements, 'Delete-Elements,
                                    'Make-Seq-Empty, 'Finish-Editing])
    let (Choice : integer       = 0,
       Temp-Seq : seq(any-type) = Current-Seq)

      (while Choice ~= 4 do
        Choice <- Make-Menu(Choice-List, "What do you want to do?");
        if Choice = 1 then
           (format(t, "adding  elements~%");
           let  (of-type : symbol = Tell-Set-Seq-Type(attr),
                 where-to-add : integer = Size(temp-seq) + 2)

             (if empty(Temp-Seq) then
                 where-to-add <- 1
              else
                (enumerate index from 1 to size(Temp-Seq) do
                  if Tell-Set-Seq-Type(attr) in
                    {'set, 'seq, 'symbol, 'real, 'integer, 'boolean} then
                    format(t, "~d - ~s~%", index, temp-seq(index))
                  else
                    format(t, "~d - ~\\pp\\~%", index, temp-seq(index))
                );
                while where-to-add > size(temp-seq) + 1 or
                              where-to-add < 1 do
                  where-to-add <-
                        Read-Integer("What index do you want to add")
             );
             if of-type = 'integer then
```

```
            temp-seq <-  insert(temp-seq, where-to-add,
                                 Read-Integer("(an integer)"))
        elseif of-type = 'real then
            temp-seq <-  insert(temp-seq, where-to-add,
                                Read-Real("(a real)"))
        elseif of-type = 'string then
            temp-seq <-  insert(temp-seq, where-to-add,
                                Read-String("(a string)"))
        elseif of-type = 'symbol then
            temp-seq <-  insert(temp-seq, where-to-add,
                                 Read-Symbol("(a symbol)"))
        elseif of-type = 'boolean then
            temp-seq <-  insert(temp-seq, where-to-add,
                                Read-Boolean("(a boolean)"))
        elseif of-type = 'any-type then
            temp-seq <-  insert(temp-seq,where-to-add,
                                Read-Any-Type("(any-type)"))
        else % must be an object
            temp-seq <- insert(temp-seq, where-to-add,
                        Make-New-Object(Tell-Set-Seq-Binding(attr)))


      ) % end if choice = 1

    elseif Choice = 2 then
       if empty(Temp-Seq) then
         format(t, "No elements to delete~%")
       else
         format(debug-on, "deleting elements~%");

         (while ~empty(Temp-Seq) and-then
                     Read-Yes-Or-No( "Delete another element? " ) do
          if Tell-Set-Seq-Type(attr) in
                {'set, 'seq, 'symbol, 'real, 'integer, 'boolean} then
             let (One-To-Delete : integer = Make-Menu(
                                 Temp-Seq,
                                "Which one do you want to delete?"))
             Temp-Seq <- delete(Temp-Seq, One-To-Delete)
          else
             let (One-To-Delete : integer = Make-Object-Menu(
                                 Temp-Seq,
                                "Which one do you want to delete?"))
             Temp-Seq <- delete(Temp-Seq, One-To-Delete)

         ) % end while

    elseif Choice = 3 then
       format(t, "making seq empty~%");
        temp-seq <- []

);
```

```
   temp-seq

%-------------------------------------------------------------------
" Reads in a group of items of the given type and puts them into a
  sequence.  Since a  sequence may already exist, it first asks if
  the user wants to change the original sequence"

function Read-Seq(attr : re::binding,
           current-seq : seq(any-type)) : seq(any-type) =

  if ~empty(current-seq) then
     Edit-Seq(attr, current-seq)
  else
    let (temp-seq : seq(any-type) = [],
         of-type  : symbol = Tell-Set-Seq-Type(attr))

      format(t, "creating a seq of type ~s ~%", of-type);
      (while Read-Yes-Or-No("Add another element? ") do

        if of-type = 'integer then
          temp-seq <-  append(temp-seq, Read-Integer("(an integer)"))
        elseif of-type = 'real then
          temp-seq <-  append(temp-seq, Read-Real("(a real)"))
        elseif of-type = 'string then
          temp-seq <-  append(temp-seq, Read-String("(a string)"))
        elseif of-type = 'symbol then
          temp-seq <-  append(temp-seq, Read-Symbol("(a symbol)"))
        elseif of-type = 'boolean then
          temp-seq <-  append(temp-seq, Read-Boolean("(a boolean)"))
        elseif of-type = 'any-type then
          temp-seq <-  append(temp-seq, Read-Any-Type("(any-type)"))
        else % must be an object
          temp-seq <- append(temp-seq,
                      Make-New-Object(Tell-Set-Seq-Binding(attr)))

      ); %end while
      temp-seq



%%%------------------ Functions for adding new objects-------------------
%-------------------------------------------------------------------
" The new object type must be a subclass of component-obj."

function Is-Valid-New-Type (Obj-Type : symbol) =
    Find-Object-Class(Obj-Type) in
     Class-Subclasses(Find-Object-Class('component-obj), true)

%-------------------------------------------------------------------
"  Asks for the name of the application for which the new object is to
   be a part, if  the application exists, it then asks for a new object
```

```
   name. It checks that the object name does not exist.  It then asks
   for the type  of object to be built,  if it is a valid type, it builds
   a new object, gets the data, and assigns it to the  application."

function Add-Object (X : object) =
  let (Applic-Name : symbol = Read-Symbol-Default(
                              "Enter the application name",
                               name(x)))

    if defined?(find-object('spec-obj, applic-name)) then
       let (obj-type : symbol =
                Read-Symbol("What type of object do you want to create?"))

         if Is-Valid-New-Type(Obj-Type) then
           let (Obj-Name : symbol =
                           Read-Symbol("What is the object's name?"))

             if  undefined?(find-object('component-obj, Obj-Name)) then
                let (New-Obj : object =
                                    Modify-Object(make-object(Obj-Type)))
                   Changes-Made <- true;
                   Set-Attrs(New-Obj, 'name, Obj-Name);
                   Set-To-Parent(New-Obj,
                                 find-object('spec-obj, Applic-name))
             else
               format(t, "An object named ~s already exists~%", Obj-Name)

         else
           format(t, "~s is not a valid object type~%", obj-type)

    else
      format(t, "Application ~s does not exist in the object base.~%",
                applic-name)



%%%------------------- Function for erasing objects---------------------
%-----------------------------------------------------------------------
" Asks for the object to be deleted, checks that the object exists in the
  object base, then asks if the user is sure he wants to erase that
  object, if he answers yes, the object is removed from the application
  definition, and erased"

function Delete-Object (A : object) =
  let (Obj-Name : symbol =
                Read-Symbol-Default("What object do you want to delete?",
                                    name(A)))
    let (Obj : object = Find-Object('component-obj, Obj-Name))
       if defined?(Obj) then
         (if Read-Yes-Or-No(concat("Are you sure you want to detete ",
                                   Symbol-To-String(Obj-Name), " " )) then
```

```
        Changes-Made <- true;
        Remove-From-Parent(Obj, Parent-Expr(Obj));
        erase-Object(Obj)
      )
else
  format(t, "~s is not a component in the object base~%", Obj-Name)
```

```
!! in-package("RU")
!! in-grammar('user)

#||
 File name: complete.re

 Description: Contains the functions to complete an application definition

 Rules:
  Complete-Application-Definition

 Functions:
  Complete-Definition
  Load-Obj-Ok
  Complete-Load-Obj
  Generic-Exists
  Generic-Ok
  Complete-Generic-Obj
  Do-Complete-Generic
  Incomplete-Obj-OK
  Complete-Incomplete-Obj
  Completely-Defined

||#


%-----------------------------------------------------------------------
rule Complete-Application-Definition(X : object)
 true --> Complete-Definition(X)


%-----------------------------------------------------------------------
"Goes through all of the objects in an application definition, and if they
 need further information from the user, prompts for the input.  The
 object classes that need this additional data are:  Load-obj (load
 a specific object instance), Generic-Inst (instantiate a generic), or
 Incomplete-Obj (incomplete definition, contains only the name and
 the object type"

function Complete-Definition (Applic : Object) =
  let (Applic-Name : symbol = Read-Symbol-Default(
                                "Enter the application name",
                                 name(Applic)))
    let (Applic-Obj : object = Find-Object('Spec-Obj, Applic-Name))
      if defined?(Applic-Obj) then
        enumerate Component over
                        Spec-Parts(Find-Object('spec-Obj, Applic-Name)) do

          (if Load-Obj(Component) then
            Complete-Load-Obj(Component, Applic-Obj)
          elseif Generic-Inst(Component) then
```

```
          Complete-Generic-Obj(Component, Applic-Obj)
        elseif Incomplete-Obj(Component) then
          Complete-Incomplete-Obj(Component, Applic-Obj) )
    else
      format(t, "Application ~s does not exist~%", Applic-Name)



%%%----------------------------Complete Load----------------------------
%---------------------------------------------------------------------
" Tests if the load-obj is valid.  To be valid, it must refer to an object
  in the technology base (the file must exist) and there is not already an
  object by that name in the object base."

function Load-Obj-OK(Obj-Name, Path-Name : string) : Boolean =

    let (Load-OK : boolean = true,
       File-Name : string = concat(Object-Path,
                                   Path-Name,
                                   lisp::string-upcase(Obj-Name),
                                   Saved-Suffix) )

      (if ~File-Exists(File-Name) then
        format(t, "~s does not exist in the technology base~%",
                    concat(Path-Name, Obj-Name));
        Load-Ok <- false
      );
      (if defined?(find-object('component-obj,
            string-to-symbol(lisp::string-upcase(Obj-Name), "RU"))) then
        format(t, "~s already exists in the object base~%", Obj-Name);
        Load-Ok <- false
      );
      Load-Ok



%---------------------------------------------------------------------
" Takes a load-obj and converts it to a 'real' object.  Checks that the
  load-obj is valid, if so it calls Load-Obj to actually load it in the
  object base, and erases the load-obj.  (Load-Obj assigns it to the
  application object).  If the object to be loaded does not exist in the
  technology base, the system will prompt the user for another name."

function Complete-Load-Obj(Load : Load-Obj, Applic-Obj : Spec-Obj) =
  let (Obj-Name : string = File-Name-Of (Object-To-Load(Load)),
       Path-Name : string = Path-Name-Of (Object-To-Load(Load)))

       if Load-Obj-Ok(Obj-Name, Path-Name) then
         Load-The-Object(Applic-Obj,  Path-Name,
                    string-to-symbol(lisp::string-upcase(Obj-Name), "RU"));
```

```
        Erase-Object(Load)
      else
       let (Replace? : boolean =
             Read-Yes-Or-No("Do you want to specify a new object name?"))
         (while Replace? and ~Load-Obj-OK(Obj-Name, Path-Name)  do
           set-attrs(Load, 'Object-To-Load,
                                  Read-Symbol("Enter the new name"));
             Obj-Name <-  File-Name-Of(Object-To-Load(Load));
             Path-Name <- Path-Name-Of(Object-To-Load(Load));
           if ~Load-Obj-OK(Obj-Name, Path-Name) then
              Replace? <-
                Read-Yes-Or-No(
                    "Do you want to specify a new object name?")
         );
         if Replace? then % succesfully replaced the value
            Load-The-Object(Applic-Obj, Path-Name,
                   string-to-symbol(lisp::string-upcase(Obj-Name), "RU"));
            Erase-Object(I ad)
        else
           Erase-Object(Load)


%%%-------------------------Complete Generics-------------------------
%-------------------------------------------------------------------------
"Since the path name may be prepended to the file name, need to strip the
 path off"

function Generic-Exists(Gen-Name : string) : boolean =

    defined?(Find-Object('Generic-Obj, string-to-symbol(Gen-Name, "RU")))

%-------------------------------------------------------------------------
" Checks that the generic-inst object is valid.  The generic template must
  exist in the object base (it was loaded before this function was
  called), the generic-inst and the generic template must have the
  same number of parameters, and finally, the types of the parameters
  must match"

function Generic-OK(Generic : Generic-Inst) : Boolean =
 let (Gen-OK : boolean = true,
      Gen-Name : string = File-Name-Of(Generic-To-Be-Used(Generic)))

  if ~Generic-Exists(Gen-Name) then
        Gen-OK <- false
  else
    let (Generic-Template : Generic-Obj =
           Find-Object('Generic-Obj, string-to-symbol(Gen-Name, "RU")))

    (if size(Generic-Parameters(Generic)) ~=
              size(Placeholder-IDs(Generic-Template)) then
```

```
   format(t, "Wrong number of parameters, should have ~d,
              but have ~d~%",
             size(Placeholder-IDs(Generic-Template)),
             size(Generic-Parameters(Generic)));
   Gen-OK <- false

else
  enumerate index from 1 to size(Placeholder-IDs(Generic-Template)) do
    let (id-type : symbol = Placeholder-Type(Generic-Template)(index),
         Gen-Parm : any-type = Generic-Parameters(Generic)(index))
      if id-type = 'symbol then
        (if ~lisp::symbolp(Gen-Parm) then
          format(t, "Invalid parameter ~s, should be a(n) ~s~%",
                    Gen-Parm, id-type);
          Gen-OK <- false
        )

      elseif id-type = 'an-object then
       (if undefined? (Find-Object ('Component-Obj, Gen-Parm)) then
          format(t,
          "~S should be an object, but it is not in the object base~%",
            Gen-Parm);
          Gen-Ok <- False
       )

      elseif id-type = 'string then
        (if ~lisp::stringp(Gen-Parm) then
          format(t, "Invalid parameter ~s, should be a(n) ~s~%",
                    Gen-Parm, id-type);
          Gen-OK <- false
        )

      elseif id-type = 'integer then
        (if ~lisp::integerp(Gen-Parm) then
          format(t, "Invalid parameter ~s, should be a(n) ~s~%",
                    Gen-Parm, id-type);
          Gen-OK <- false
        )

      elseif id-type = 'real then
        (if ~lisp::floatp(Gen-Parm) then
          format(t, "Invalid parameter ~s, should be a(n) ~s~%",
                    Gen-Parm, id-type);
          Gen-OK <- false
        )

      elseif id-type = 'boolean then
        (if Gen-Parm ~= 'true and Gen-Parm ~= 'nil then
          format(t, "Invalid parameter ~s, should be a(n) ~s~%",
                    Gen-Parm, id-type);
          Gen-OK <- false
```

```
          )
        % else  id-type = 'any-type then, it's ok
   );
   Gen-Ok

%-------------------------------------------------------------------------
" Instantiates the generic, assigns it to the application definition, and
  erases the generic-inst."

function Do-Complete-Generic(Generic :  Generic-Inst,
                         Applic-Obj : Spec-Obj) =

  let (New-Obj-name : symbol =
          string-to-symbol(File-Name-Of(Name(Generic)), "RU"))
    Instantiate-Generic(Generic);
    Remove-From-Parent(Generic, Applic-Obj);
    Set-To-Parent(Find-Object('component-obj, New-Obj-Name), Applic-Obj);

    (enumerate New-Obj over Obj-List(Find-Object('Generic-Obj,
              String-to-Symbol(File-Name-Of(
                  Generic-To-Be-Used(Generic)), "RU"))) do
      Set-To-Parent(Find-Object('Component-Obj, New-Obj), Applic-Obj)
    );
    Erase-Object(Generic)


%-------------------------------------------------------------------------
" Given a generic-inst (what the application specialist enters to
  instantiate  a generic and the application definition object, loads
  the generic from the object base, checks that the generic-inst is ok,
  and completes it.  If the generic is invalid, the application
  specialist does get a second chance, and the generic-inst is deleted."

function Complete-Generic-Obj(Generic : Generic-Inst,
                          Applic-Obj : Spec-Obj) =

  Load-A-Generic(Generic-To-Be-Used(Generic));
  if Generic-OK(Generic) then
    format(t, "Instantiating Generic ~\\pp\\~%", Generic);
    Do-Complete-Generic(Generic, Applic-Obj)

  else
   format(t, "Invalid generic object ~\\pp\\~%", generic);
   % give the option to replace the name and/or object type
   (let (Replace? : boolean =
                     Read-Yes-Or-No("Do you want to edit the object?"))
      (while Replace? and ~Generic-OK(Generic) do
         (if Generic-Exists(
                File-Name-Of(Generic-To-Be-Used(Generic))) then
           Generic <-
                Edit-Generic-Inst(Generic, Generic-To-Be-Used(Generic));
```

```
                        % doesn't allow you to edit the name
          Set-Attrs(Generic, 'name, Read-Symbol-Default("Enter the name",
                                                        name(generic)))
        else
          Set-Attrs(Generic, 'Generic-To-Be-Used,
                            Read-Symbol("Enter generic to use"));
          Load-A-Generic(Generic-To-Be-Used(Generic))
        );
        if ~Generic-OK(Generic) then %editing didn't fix the problems
          Replace? <-  Read-Yes-Or-No("Do you want to edit the object?")
      );
      if Replace? then
      % successfully edited the object (otherwise, replace will be false)
        Do-Complete-Generic(Generic, Applic-Obj)
      else
       Remove-From-Parent(Generic, Applic-Obj); % Remove from application
       Erase-Object(Generic)
    )

%%%------------------------Complete Incomplete-------------------------
%--------------------------------------------------------------------------
"Checks that the incomplete object is ok.  (This probably doesn't really
 need to be a separate function, but this is consistent with the
 other two"

function Incomplete-Obj-OK(Incomp : Incomplete-Obj) : boolean =
  undefined?(Find-Object('component-obj, name(incomp))) &
     Is-Valid-New-Type(Obj-Type(Incomp))   % from modify routines

%--------------------------------------------------------------------------
" If the incomplete-obj is valid, creates a new object of the appropriate
  type, gives it the new name, assigns it to the application definition,
  gets all of the attribute information, then erases the incomplete-obj.
  If the object class is not valid, it asks the application specialist if
  he wants to change the object type or the name"

function Complete-Incomplete-Obj(Incomp : Incomplete-Obj,
                                 Applic-Obj : Spec-Obj) =

  format(t, "completing an incomplete object~%");

  if Incomplete-Obj-Ok(Incomp) then
     let (new-obj : object = make-object(obj-type(Incomp)))
       Set-Attrs(new-obj, 'name, name(Incomp));
       Set-To-Parent(new-obj, Applic-Obj);
       format(t, "Complete the information for object ~S:~%",
                  name(incomp));
       New-Obj <- Modify-Object(new-obj);
       Erase-Object(Incomp)

  else
```

```
   format(t, "Invalid object  ~\\pp\\ ~%", Incomp);
     % give the option to replace the name and/or object type
     (let (Replace? : boolean =
               Read-Yes-Or-No("Do you want to edit the object?"))
       (while Replace? and ~Incomplete-Obj-OK(Incomp) do
         Incomp <- Modify-Object(Incomp);
            % doesn't allow you to edit the name
         Set-Attrs(Incomp, 'name,
                  Read-Symbol-Default("Enter the name", name(incomp)));
         if ~Incomplete-Obj-Ok(Incomp) then
                % editing didn't fix the problem(s)
           Replace? <-  Read-Yes-Or-No("Do you want to edit the object?")
       );
       if Replace? then
       % successfully edited the object (otherwise, replace will be false)
         let (new-obj : object = make-object(obj-type(Incomp)))
           Set-Attrs(new-obj, 'name, name(Incomp));
           Set-To-Parent(new-obj, Applic-Obj);
           New-Obj <- Modify-Object(new-obj);
           Erase-Object(Incomp)
       else
        Erase-Object(Incomp)
      )

%-------------------------------------------------------------------------
" Tells if the application definition is completely defined (generic
  objects instantiated, load objects loaded, and incomplete objects
  completed"

function Completely-Defined(Applic : Spec-Obj) : boolean =
  ~ex (obj) ( obj in Spec-Parts(Applic) &
            (load-obj(obj) or Incomplete-Obj(obj) or Generic-Inst(obj)) )
```

```
!! in-package("RU")
!! in-grammar('user)

#||
 File name: generic.re

 Description: Contains the functions necessary to instantiate a genric
  object, to add generic instances to the application definition, to
  edit the generic instances already in the application defintion, and
  finally, to load the generic object into the object base.

 Rules:
  Add-Generic-Instance

 Functions:
  Replace-Values
  Replace-In-Set
  Replace-In-Seq
  Load-A-Generic
  Instantiate-Generic
  Make-Generic-Inst
  Build-Generic-Inst
  Edit-Generic-Inst

||#

%-----------------------------------------------------------------------
rule Add-Generic-Instance(Applic)
  true --> Build-Generic-Inst(Applic)

%-----------------------------------------------------------------------
" Goes through all of the attributes in the object looking for those whose
  value is the same as old-value.  When it finds an occurance, it repalces
  the attribute with new value.  It replaces all of the occurances of
  old-value with new-value."

function Replace-Values(In-Object : object,
             Old-Value, New-Value : any-type) =


  let (attr-list : set(re::binding) =  Return-Attribute-List(In-Object))
    (let (attr-with-values : set(re::binding) =
             {atrs | (atrs) atrs in attr-list &
                 equal( Retrieve-Attribute(In-Object, atrs), Old-Value) })
    % NOTE: using the lisp equal function so it compares strings correctly

      enumerate atr-to-change over attr-with-values do
         Store-Attribute (In-Object, atr-to-change, New-Value)
     );

     % if attribute is a set  look for each member of the set
```

```
    let (set-attr-list : set(re::binding) =
          {atrs | (atrs) atrs in attr-list & Tell-Type(atrs) = 'set})

      (enumerate set-attr over set-attr-list do
        if defined? (Find-Object-Class(Tell-Set-Seq-Type(Set-Attr))) then
          %its an object
          enumerate set-member over
                              Retrieve-Attribute(In-Object, set-attr) do
            Replace-Values(Set-member, Old-Value, New-Value)
        else
            Store-Attribute(In-Object, Set-Attr,
                Replace-In-Set(Retrieve-Attribute(In-Object, set-attr),
                               Old-Value, New-Value))
      );

    % if attribute is a seq look for each member of the seq
    let (seq-attr-list : set(re::binding) =
        {atrs | (atrs) atrs in attr-list & Tell-Type(atrs) = 'seq })

      (enumerate seq-attr over seq-attr-list do

        if defined?( Find-Object-Class(Tell-Set-Seq-Type(Seq-Attr))) then
          enumerate seq-member over
                              Retrieve-Attribute(In-Object, seq-attr) do
            Replace-Values(Seq-member, Old-Value, New-Value)
        else
            Store-Attribute(In-Object, Seq-Attr,
                  Replace-In-Seq(Retrieve-Attribute(In-Object, seq-attr),
                                 Old-Value, New-Value))
      );


    % now look for possible object attributes that need to
    % have values replaced
    let (obj-attr-list : set(re::binding) =
          {atrs | (atrs) atrs in attr-list & Tell-Type(atrs) = 'object})
      enumerate obj-attr over obj-attr-list do
        Replace-Values(Retrieve-Attribute(In-Object, Obj-Attr),
                       Old-Value, New-Value)


%-------------------------------------------------------------------------
" Replaces an occurance of old-value in a set with new-value"

function Replace-In-Set(The-Set : set(any-type),
           Old-Value, New-Value : any-type) : set(any-type) =

  let (New-Set : set(any-type) = The-Set)
    New-Set <- New-Set less Old-Value;
    New-Set <- New-set with New-Value;
    New-Set
```

```
%-------------------------------------------------------------------------
" Replaces all occurances of old-value with new-value in a sequence"

function Replace-In-Seq(The-Seq : seq(any-type),
          Old-Value, New-Value : any-type) : seq(any-type) =

  let (New-Seq : seq(any-type) = The-Seq)
    (enumerate index from 1 to size(The-Seq) do
      if The-Seq(index) = Old-Value then
         New-Seq(index) <- New-Value
    );
    New-Seq


%-------------------------------------------------------------------------
" Given a generic instance to instantiate, it creates a new object and
  replaces all of the placeholders with the new values.  Assumes the
  generic has already been loaded"

function Instantiate-Generic(Generic : generic-inst) =
 let (Obj-Name : string = File-Name-Of(Generic-to-be-used(Generic)))

   let (generic-object =
          find-object('generic-obj, string-to-symbol(Obj-Name, "RU")),
        save-name = name(Generic))

     set-attrs(Generic, 'name, newsymbol(name(generic))); %change the name
     let (new-object : object = copy-object(
                   find-object('component-obj,
                                obj-instance(Generic-Object))))
       set-attrs(New-Object, 'name, save-name);
       (enumerate index from 1 to
                  size(Placeholder-IDs(Generic-Object)) do
          Replace-Values(New-Object,
                         Placeholder-IDs(Generic-Object)(index),
                         Generic-Parameters(Generic)(index))
      );
     % if any of the import areas were assigned to internal outputs,
     % they refer to the generic name, not the new object name, so replace
     % the generic name with the new object name
     Replace-Values(New-Object, Obj-Instance(Generic-Object), save-name)


%-------------------------------------------------------------------------
" Loads a generic into the object base.  The name of the generic should
  match the name of the file.  It is assumed that all generics are stored
  in the same path."

function Load-A-Generic(Gen-Name : symbol) =
  let (File-Name : string =
```

```
                (concat(Generics-Path, symbol-to-string(Gen-Name),
                        Saved-Suffix)))
    if File-Exists(File-Name) then
      Parse-In-File(File-Name)
    else
      format(t, "The generic ~s does not exist in the technology base ~%",
                gen-name)


%-------------------------------------------------------------------------
" Actually makes the generic instance."

function Make-Generic-Inst(Generic-Template : symbol) : Generic-Inst =
 let (New-Inst-Name : symbol =
       Read-Symbol("Enter the name for the generic instance"))
    if undefined?(Find-Object('world-obj, New-Inst-Name)) then
      let (New-Inst : Generic-Inst =
               Edit-Generic-Inst(make-object('Generic-Inst),
                                 Generic-Template) )
        set-attrs(New-Inst, 'name, New-Inst-Name,
                            'Generic-To-Be-Used, Generic-Template);
        New-Inst % return the new instance
    else
      format(t, "Sorry, there's already a component named ~s~%",
                 New-Inst-Name);
      re::*undefined*  %can't make an instance


%-------------------------------------------------------------------------
" Allows the user to add a generic instance to the application definition.
  Asks for the application definition name and the generic name.  It loads
  the generic (if it exists), calls make-generic-inst, and finally,
  assigns it to the application definition."

function Build-Generic-Inst(For-Application) =
  let (Path-Name = lisp::string-upcase(
       Read-String(concat("Enter the path name (include last /)",
                          Generics-Path))))
    format(t, "Here are the existing generics:~%");
    Display-Files(concat(Generics-Path, Path-Name));

    let (Generic-Template  : symbol = string-to-symbol(concat(Path-Name,
         lisp::string-upcase(Read-String(
             "Enter the name of the generic to instantiate"))), "RU"),
       Applic-Name  : symbol = 'dummy)

      if File-Exists(concat(Generics-Path,
                            symbol-to-string(Generic-Template),
                            Saved-Suffix)) then

       Load-A-Generic(Generic-Template);
```

```
      (if Spec-Obj(For-Application) then
          % get the name of the application
         Applic-Name <- Read-Symbol-Default("Enter the application name",
                                          name(For-Application))
      else
        Applic-Name <- Read-Symbol("Enter the application name")
      );

      (if defined?(find-object('Spec-Obj, Applic-Name)) then
                  % have a valid application name
        let (New-Inst : Generic-Inst =
                 Make-Generic-Inst (Generic-Template))
          if defined?(New-Inst) then
            Set-To-Parent(New-Inst, Find-object('Spec-Obj, Applic-Name))
          else
            format(t, "Application ~s does not exist~%", Applic-Name)
      )
     else
       Format(t, "Object ~s does not exist in the technology base~%",
              Generic-Template)


%-------------------------------------------------------------------------
" Given a generic-inst, allows the user to modify all the attributes.
  If the object is just being created, it must initialize the
  generic-parameters sequence to be the same length as the parameters
  in the generic template."

function Edit-Generic-Inst(Gen-Inst : Generic-Inst,
                   Generic-Template : symbol) : object =

   (let(Generic  : Generic-Obj  = Find-Object('Generic-Obj,
             string-to-symbol(File-Name-Of(Generic-Template), "RU") ) )

      enumerate index from 1 to size(Placeholder-IDs(Generic)) do
        let (Parm-Type : symbol   = Placeholder-Type(Generic)(index),
           Placeholder : any-type = Placeholder-IDs(Generic)(index),
           Current-Val : any-type = Generic-Parameters(Gen-Inst)(index))
           (if empty(Generic-Parameters(Gen-Inst)) then
               % it's a new instance
              enumerate index1 from 1 to size(Placeholder-IDs(Generic)) do
                Generic-Parameters(Gen-Inst) <-
                    append( Generic-Parameters(Gen-Inst),
                            re::*undefined* ) % initialize seq
           );
           % if there aren't enough parameters, pad the sequence
           (while size(Generic-Parameters(Gen-Inst)) <
                      size(Placeholder-IDs(Generic)) do
                Generic-Parameters(Gen-Inst) <-
                      append( Generic-Parameters(Gen-Inst),
                              re::*undefined* ) %
```

```
        );

       % if there are too many parameters, truncate
       Generic-Parameters(Gen-Inst) <-
            subseq(Generic-Parameters(Gen-Inst), 1,
                   size(Placeholder-IDs(Generic)));
       if Parm-Type = 'symbol or Parm-Type = 'an-object then
          Generic-Parameters(Gen-Inst)(index) <- Read-Symbol-Default(
                  concat("Enter a symbol for ",
                         symbol-to-string(Placeholder)), Current-Val)

       elseif Parm-Type = 'string then
         Generic-Parameters(Gen-Inst)(index) <- Read-String-Default(
               concat("Enter a string for ", Placeholder), Current-Val)

       elseif Parm-Type = 'integer then
          Generic-Parameters(Gen-Inst)(index) <- Read-Integer-Default(
                concat("Enter an integer for ",
                           lisp::princ-to-string(Placeholder)),
                           Current-Val)

       elseif Parm-Type = 'real then
          Generic-Parameters(Gen-Inst)(index) <-  Read-Real-Default(
                concat("Enter a real for ",
                      lisp::princ-to-string(Placeholder)), Current-Val)

       elseif Parm-Type = 'boolean then
          Generic-Parameters(Gen-Inst)(index) <- Read-Boolean-Default(
                concat("Enter a symbol for ",
                         symbol-to-string(Placeholder)), Current-Val)

       else % Parm-Type = 'any-type then
          Generic-Parameters(Gen-Inst)(index) <-  Read-Symbol-Default(
                concat("Enter any-type for ",  Placeholder),
                Current-Val)

);
Gen-Inst % return the new and improved generic instance
```

```
!! in-package("RU")
!! in-grammar('user)

#||
 File name: build-generic.re

 Description: This file is to help the software engineer create
              generic objects

 Rules:
  Build-Generic
  Save-Generic

 Functions:
  Build-A-Generic
  Save-A-Generic


||#

%-----------------------------------------------------------------------
 rule Build-Generic(Gen : object)
  true --> Build-A-Generic()

 rule Save-Generic(Gen : object)
   true --> Save-A-Generic(Gen)

%-----------------------------------------------------------------------
" Aids the software engineer in building generics.  The software engineer
  must first create an instance of an object, including the placeholders.
  The name of this object is the obj-instance attribute for the generic
  template.  Then the system prompts the software engineer for the
  placeholder type and then asks for a value of that type. "

function Build-A-Generic() =

  let (Obj-Inst : symbol =
           Read-Symbol("Enter the name of the object instance"),
       Generic-Name : symbol =
           Read-Symbol("Enter the name of the generic"),
       generic : generic-obj = make-object('generic-obj))

    if defined?(Find-Object('Generic-Obj, Generic-Name)) then
       format(t, "Generic ~s already exists~%", Generic-Name);
       erase-object(generic)

    elseif undefined?(Find-object('component-obj, Obj-Inst)) then
      format(t, "Object ~s doesn't exist~%", obj-inst)
    else
      format(t, "assigning instance to ~\\pp\\ ~%",
                          find-object('component-obj, obj-inst));
```

```
set-attrs(generic, 'name, Generic-name);
set-attrs(generic, 'obj-instance, Obj-Inst);

(let (ID-List   : seq(any-type) = [],
      Type-List : seq(symbol) = [],
      Place-Type : symbol = 'integer)

  (while  Read-Yes-Or-No
        ("Do you want to add more generic parameters?") do
    format(t, "Valid types are: ~s and ~s~%",
              Valid-Types, 'an-object);
    Place-Type <- Read-Symbol("Enter it's type");
    if (Place-Type in Valid-Types) or Place-Type = 'an-object  then

     let (fun-name : symbol =
          string-to-symbol(lisp::string-upcase(concat("Read-",
                           symbol-to-string(Place-Type))), "RU"))

        (if Place-Type = 'an-object then
          (let (Place : symbol =
                 Read-Symbol( "Enter the placeholder"))
             if Valid-Placeholder(
                       Find-Object('Component-obj, obj-inst),
                       Place, Place-Type) then
               ID-List <- append(ID-List, Place)
             else
               format(t, "Invalid placeholder~%")
          )
        else
          (let (Place : symbol =
                 funcall( fun-name, "Enter the placeholder"))
             if Valid-Placeholder(
                       Find-Object('Component-obj, obj-inst),
                       Place, Place-Type) then
               ID-List <- append(ID-List, Place)
             else
               format(t, "Invalid placeholder~%")
          )
        );
       Type-List <- append(Type-List, Place-Type)

    else
     format(t, "~s is an invalid type in this system~%", Place-Type)

  );
  set-attrs(generic, 'placeholder-ids, ID-List);
  set-attrs(generic, 'placeholder-type, Type-List)
);
Format(t, "What objects are needed to go with this generic?~%");
while Read-Yes-Or-No("Add another object ") do
  let (Obj-to-add : symbol =
```

```
            Read-Symbol("Enter the name of the object"))
        defined?(Find-Object('component-obj, Obj-To-Add)) -->
             Obj-to-add in Obj-List(Generic);
        undefined?(Find-Object('component-obj, Obj-To-Add)) -->
             format(t, "Object ~s does not exist,
                        cannot add to object list~%",
                        Obj-To-Add)


%-----------------------------------------------------------------------
" Saves a generic to a file.  The file name will match the name of the
  generic.  The generic is saved in two parts, the generic object itself,
  and the object instance."

 function Save-A-Generic(Gen : object) =
   let (Obj-Name : symbol =
          Read-Symbol-Default("Enter generic object to save", name(Gen)))
     let (Gen-Obj : Generic-Obj = find-Object('Generic-Obj, Obj-Name),
        File-Name : string =  concat(Generics-Path,
                 lisp::string-upcase(Read-String(
                           concat("Enter the path (include last /) ",
                      Generics-Path))),
                 symbol-to-string(Obj-Name),
                 Saved-Suffix))
       if File-Exists(File-Name) then
         format(t, "A generic called ~s has already been saved~%",
                   Obj-Name)
       else
         if undefined?(Gen-Obj) then
           format(t, "Object ~s does not exist~%", Obj-Name)
         elseif undefined?(
                find-object('component-obj, Obj-Instance(Gen-Obj))) then
           format(t, "The object instance ~s does not exist in
                      the object base~%",
                      Obj-Instance(Gen-Obj))
           elseif Read-Yes-Or-No(concat("Do you really want to save ",
                     File-name, " ")) then  % save the object to a file

             rd-on(File-name); %redirect standard output to file
             format(t, "~\\pp\\ ~%", Gen-Obj);
             format(t, "~\\pp\\",  find-object('component-obj,
                                               Obj-Instance(Gen-Obj)));
             format(t, "~%");
             (enumerate obj over obj-list(Gen) do
               if defined?(Find-Object('Component-Obj, obj)) then
                 format(t, "~\\pp\\~%", find-object('component-obj, obj));
                 format(t, "~%")
             );
             rd-off()
```

```
%-----------------------------------------------------------------------
"Checks to make sure that the placeholder actually exists in the
 object instance"

function Valid-Placeholder(Obj : object, Placeholder : any-type,
                                          Place-Type : symbol) =

  let (OK : boolean = false,
       attr-list : set(re::binding) =  Return-Attribute-List(Obj))
    (let (attr-with-values : set(re::binding) =
                  {atrs | (atrs) atrs in attr-list &
                  equal( Retrieve-Attribute(Obj, atrs), Placeholder) })
    % NOTE: using the lisp equal function so it compares strings correctly

     if ~empty(attr-with-values) then
         OK <- true
     else

       % if attribute is a set  look for each member of the set
       (let (set-attr-list : set(re::binding) =
            {atrs | (atrs) atrs in attr-list &
            Tell-Type(atrs) = 'set})

        enumerate set-attr over set-attr-list do
         if defined?(Find-Object-Class(Tell-Set-Seq-Type(Set-Attr))) then
               %its an object
          (enumerate set-member over Retrieve-Attribute(Obj, set-attr) do
            if Valid-Placeholder(Set-Member, Placeholder, Place-Type) then
              Ok <- true
          )
         else
           OK <- Placeholder in Retrieve-Attribute(Obj, set-attr)
       );

     if ~OK then
       % if attribute is a seq look for each member of the seq
      (let (seq-attr-list: set(re::binding) =
          {atrs | (atrs) atrs in attr-list &
          Tell-Type(atrs) = 'seq })

       enumerate seq-attr over seq-attr-list do
         if defined?( Find-Object-Class(Tell-Set-Seq-Type(Seq-Attr))) then
           (enumerate seq-member over Retrieve-Attribute(Obj, seq-attr) do

            if Valid-Placeholder(Seq-member, Placeholder, Place-Type) then
                OK <- true
           )
         else
           OK <- Placeholder in Retrieve-Attribute(Obj, seq-attr)

       );
```

```
   if ~OK then
   % now look for possible object attributes that need to have
   % values replaced
   (let (obj-attr-list : set(re::binding) =
             {atrs | (atrs) atrs in attr-list &
                              Tell-Type(atrs) = 'object})
     enumerate obj-attr over obj-attr-list do
      Valid-Placeholder(Retrieve-Attribute(Obj, Obj-Attr),
                        Placeholder, Place-Type)


   )
);

OK
```

```
!! in-package("RU")
!! in-grammar('user)

#||
 File name: Display-Files

 Description: Given a path, displays the files in that path that end
              with the saved-suffix

 Rules:
  None

 Functions:
  Display-Files
  Path-Name-Of
  File-Name-Of

||#


%-----------------------------------------------------------------------
" Displays all the file names in the given path that have the saved-suffix
  ending"

function Display-Files (Path-name : string) =
  let (file-list : set(any-type) = lisp::Directory(path-name))
    enumerate f over file-list do
       let (f-name : string = arb( {s | (s:string)
                    pathname-name(f) = [$s, $Saved-Suffix]}))
        format(defined?(f-name), "~s~%", f-name)

%-----------------------------------------------------------------------
" Given a file name with it's path, returns just the file name portion"

function File-Name-Of(Full-Path : symbol) : string =

    if #\/ ~in symbol-to-string(Full-Path) then
      % no path name, return input
      symbol-to-string(Full-Path)
    else
       arb ( {s | (s : string)
           symbol-to-string(Full-Path) =  [..,#\/,  $s] & #\/ ~in s } )

%-----------------------------------------------------------------------
" Given a file name with it's path, returns just the path name portion"

function Path-Name-Of(Full-Path : symbol) : string =

    if #\/ ~in symbol-to-string(Full-Path) then
      % no path name, just return ""
      ""
```

```
else
  let(filenm : string = File-Name-Of(Full-Path))
    arb( {s | (s : string) symbol-to-string(Full-Path) =
                              [$s, $filenm] })
```

```
!! in-package("RU")
!! in-grammar('user)

#||
 File name: erase.re

 Description: Erases an application definition

 Rules:
  Erase-Rule

 Functions:
  Erase-Objs
||#
%-----------------------------------------------------------------------
rule Erase-Application-Definition(x : spec-obj)
   true
 -->
   Erase-Objs(x)

%-----------------------------------------------------------------------
" Asks the user which application definition to erase, the erases all
  objects in the object base that have that application as an ancestor"

function Erase-Objs(X: Spec-Obj) =
  let (applic-to-erase : symbol =
       Read-Symbol-Default(
           "Which application definition do you want to erase?", name(x)))

    let (applic : Spec-Obj = Find-Object('Spec-Obj, Applic-To-Erase))
     if defined?(Applic) then

      (enumerate Obj over {an-obj | (an-obj) world-obj(an-obj) &
                          up-to-root(an-obj) = applic} do

        format(debug-on, "Erasing object ~\\pp\\ ~%", obj);
        Erase-Object(Obj)
      );

      Erase-Object(Applic);
      format(t, "Erased application definition ~s~%", applic-to-erase)
```

```
!! in-package("RU")
!! in-grammar('user)

#||
 File name: globals.re

 Description: Contains all the global constants and variables.

||#


constant  Saved-Suffix : string = "-SAVED"

constant Generics-Path : string = "./generics/"

constant Applic-Path : string = "./applics/"

constant Object-Path : string = "./objs/"


var Fatal-Error : boolean = false

var Changes-Made : boolean = false

var Debug-On : boolean = false

var Semantic-Checks-Performed : boolean = false
```

```
!! in-package("RU")
!! in-grammar('user)

#||
 File name: menu.re

 Description: Contains functions that will query the user for a selection
  from a list of possible choices

 Functions:
  Make-Menu
  Make-Object-Menu

||#


%-------------------------------------------------------------------------
" Given a sequence of symbols and a prompt string, this function displays
  the prompt, lists all of the symbols in the sequence, and prompts the
  user for a selection.  If the selection is not in the proper range
  (i.e., between 1 and the size of the sequence), it displays an error
  message and reprompts for a selection"


function Make-Menu(Menu-Choices : seq(symbol),
                         Prompt : string) : integer =

  let (Response : integer = size(Menu-Choices) + 1)
   (while Response > size(Menu-Choices) or Response < 1 do
      format(t, "~s ~%", Prompt);
      (enumerate index from 1 to size(Menu-Choices) do
          format(t, "~d ) ~s ~%", index,  Menu-Choices(index))
       );
       Response <- Read-Integer("");
       if Response > size(Menu-Choices) or Response < 0 then
         format(t, "Invalid Response ~%")
   ); % end while
   Response

%-------------------------------------------------------------------------
" Same as above execept it takes a list of objects and uses the name
  to display the choices"

function Make-Object-Menu(Menu-Choices : seq(object),
                      Prompt : string) : integer =

  let (Response : integer = size(Menu-Choices) + 1)
   (while Response > size(Menu-Choices) or Response < 1 do
      format(t, "~s ~%", Prompt);
      (enumerate index from 1 to size(Menu-Choices) do
          if defined?(name(Menu-Choices(index))) then
```

```
        format(t, "~d ) ~s ~%", index, name(Menu-Choices(index)))
      else
        format(t, "~d ) ~\\pp\\ ~%", index, Menu-Choices(index))
    );
    Response <- Read-Integer("");
    if Response > size(Menu-Choices) or Response < 0 then
      format(t, "Invalid Response ~%")
); % end while
Response
```

```
!! in-package("RU")
!! in-grammar('user)

#||
 File name: obj-utilities.re

 Description:  This file contains functions useful when manipulating
  objects, regardless of the domain model being used.


 Rules:
  none

 Functions:
  Return-Attribute-List
  Tell-Set-Seq-Type
  Tell-Set-Seq-Binding
  Get-Attribute-List (For testing and debugging)
  Tell-Type
  Copy-Object

||#

%-----------------------------------------------------------------------
" The type-map gives a conversion from the Refine representation of
  data types to simple symbols (Note: strings are handled differently so
  they aren't in this list"

var Type-Map : map(symbol, symbol) =
  {| 're::powerset-op      -> 'set,
     're::powersequence-op -> 'seq,
     're::symbol-op        -> 'symbol,
     're::real-op          -> 'real,
     're::integer-op       -> 'integer,
     're::boolean-op       -> 'boolean,
     're::character-op     -> 'character,
     're::any-type-op      -> 'any-type|}


%-----------------------------------------------------------------------
" REFINE's attributes"
var predefined-attributes : set(symbol) =
 {'RE::--TOP-LEVEL-PREPENDUM--,
  'RE::STORED-PROPERTIES,
  'RE::BROWSER-MENU-STRING-FOR-NAMED-OBJECT--SAME-PACKAGE,
  'RE::BROWSER-MENU-STRING-FOR-NAMED-OBJECT--OTHER-PACKAGE,
  'RE::ORDERED-CHILDREN-ATTRIBUTES,
  'RE::CONSTRUCTION-FUNCTION-ATTRIBUTE-ARGS,
  'RE::CONSTRUCTION-FUNCTION,
  'RE::REFINE-INTERNAL?,
  'RE::QUOTED?,
```

```
  'RE::LISP-GETFN,
  'RE::LISP-INITIALIZE,
  'RE::LISP-FUNCTION,
  'RE::ALREADY-WARNED-ABOUT,
  'RE::SUBPART-OF,
  'RE::SUBPARTS,
  'RE::COMPILATIONS,
  'RE::ZL-DOCUMENTATION,
  'RE::USED-BY,
  'RE::MENTIONED-BY,
  'RE::DATA-TYPE,
  'RE::CHILDREN-ENVIRONMENTS,
  'RE::PARENT-ENVIRONMENT,
  'RE::COPY-OF,
  'RE::BINDING-VALUE-OF,
  'RE::PARENT-LINK-NAME,
  'RE::PARENT-LINK,
  'RE::CLASS,
  'RE::ELEMENT-OF,
  'RE::PROPS-FROM-READER}


%-------------------------------------------------------------------------
" Given an object, returns a set of bindings that represent the attributes
  removing the predefined attributes"

function Return-Attribute-List(Obj : object) : set(re::binding) =
  {atrs | (atrs) (atrs in class-attributes(instance-of(obj), true)) &
                ~(name(atrs) in predefined-attributes)}


%-------------------------------------------------------------------------
" Returns all of the subclasses of an object, NOT including the original
 object class"

function Get-SubNodes(obj-type: re::binding) : set(re::binding) =
  let (tempset : set(re::binding) =  class-subclasses(obj-type, false) )
   tempset less obj-type % don't include the original object type


%-------------------------------------------------------------------------
" This function determines the type of the set/seq attribute.  If the type
  is an object, it returns the object class name"

function Tell-Set-Seq-Type(attr : re::binding) : symbol =
  let (its-type : object = re::base(re::range-type(re::data-type(attr))))
     if re::class(its-type) = 're::binding-ref then
        re::bindingname(its-type) % returns 'string or '(object-type)
     else
       Type-Map(re::class(its-type))
```

```
%-------------------------------------------------------------------------
" Returns the type of a set or sequence as a binding (assumes its
  some object type)"

function Tell-Set-Seq-Binding(attr : re::binding) : re::binding =
   re::ref-to(re::base(re::range-type(re::data-type(attr))))


%-------------------------------------------------------------------------
" Displays all of the user-defined attributes and their types.  If want
  to see all the attributes, don't comment any lines, if only want to
  see the user-defined attributes, comment out the line: if ~(name(atr)
  in predefined-attributes) then"

function Get-attribute-list(obj : object) =
 let (attr-list : set(re::binding) =
                class-attributes(instance-of(obj), true))
  format(t, "attributes are: ~%");
  enumerate atr over attr-list do
%   if ~(name(atr) in predefined-attributes) then
      format(t, "attr: ~s type ~s ~%", atr, tell-type(atr))


%-------------------------------------------------------------------------
" Goes through the abstract syntax tree for the representation of the
  attribute to find out the attribute's data type.  Since all attributes
  are maps, we need to look at the range-type of the data-type.  Both
  objects and strings have the same representation at this level so
  there's a special test for those.  Otherwise, it uses the Type-Map to
  translate the type to a simpler-form."

function Tell-Type(attr : re::binding) : symbol =
  let (its-type : object = re::range-type(re::data-type(attr)))

    if re::class(its-type) = 're::binding-ref then
      if defined?(re::bindingname(its-type)) and-then
                  re::bindingname(its-type) = 'string then
        'string
      else
       'object
    else
      Type-Map(re::class(its-type))


%-------------------------------------------------------------------------
" makes a copy of an object, the calling routine must name the new object
 used instead of copy-term because copy-term cannot be used with unique
 names classes see Refine manual pg 3-194.  An alternative could be to
 undefine the name, then copy it.  This is a problem if the object
 contains any named objects.  This function handles any case (that I
```

```
 can think of)"

function Copy-Object(from-obj : object) : object =

  let (Attrs : set(re::binding) = Return-Attribute-List(from-obj),
       To-Obj : object = make-object(name(instance-of(from-obj))))

    (enumerate attrib over Attrs do

       % if it's an object, copy the object and assign the new
       % one to the attribute
       (if tell-type(attrib) = 'object then
          let (sub-obj : object =
               Copy-Object(retrieve-attribute(from-obj, attrib)))
               store-attribute(to-obj, attrib, sub-obj)


        elseif tell-type(attrib) = 'set then

         let (temp-set : set(any-type) = {})
          % if it is a set of objects, copy each object,
          % otherwise copy the set
          (if defined? (Find-Object-Class(Tell-Set-Seq-Type(Attrib)))
                                    then % object
            enumerate Set-Item over
                      Retrieve-Attribute(From-Obj, Attrib) do
               Temp-Set <- Temp-Set with Copy-Object(Set-Item)
           else
             Temp-Set <- Retrieve-Attribute(From-Obj, Attrib)
          );
          Store-Attribute(To-Obj, Attrib, Temp-Set)



       elseif tell-type(attrib) = 'seq then

         let (temp-seq : seq(any-type) = [])
          % if it is a seq of objects, copy each object,
          % otherwise copy the seq
          (if defined? (Find-Object-Class(tell-set-seq-type(attrib)))
                                    then % = object
            enumerate seq-item over
                      retrieve-attribute(from-obj, attrib) do
               temp-seq <- append(temp-seq, copy-object(seq-item))
           else
             temp-seq <- retrieve-attribute(from-obj, attrib)
          );
          store-attribute(to-obj, attrib, temp-seq)


        else %not a set, seq, or object
```

```
      store-attribute(to-obj, attrib,
                retrieve-attribute(from-obj, attrib))
    ) % end if
  ); %end enumerate

  To-Obj  % return the object
```

```
!! in-package("RU")
!! in-grammar('user)

#||
 File name: read-utilities.re

 Description: Contains functions that read in different data types.  They
  perfrom all type checking so the calling program is guaranteed to get a
  value of the correct type.  The read with defaults allows the calling
  program to send a default value.  If the user enters return, this value
  is returned.

 Rules:
  None

 Functions:
  Read-String
  Read-Integer
  Read-Real
  Read-Symbol
  Read-Boolean
  Read-Any-Type
  Read-Yes-Or-No
  Read-String-Default
  Read-Integer-Default
  Read-Real-Default
  Read-Symbol-Default
  Read-Boolean-Default
  Read-Any-Type-Default


||#

% what read-input returns if given a carriage return
var Null-Value  : any-type = ""


" Used to tell the valid types that can be read using these functions
  This will allow programs to test if a symbol is in Valid-Types so it
  can build the function call and use the lisp funcall call to invoke
  the proper program. This avoids big if-then-elseif statements "

var Valid-Types : set(symbol) = {'string, 'integer, 'real, 'symbol,
                                 'boolean, 'any-type}

%%------------------------------ Read Functions------------------------
% Contains functions to read in data of a specific data type.  If the
% input is not valid, it reports the error, and prompts for another
% value

%-------------------------------------------------------------------
```

```
function Read-String(Prompt : string) : string =
  (if ~empty(Prompt) then
    format(t, "~A: ", Prompt)
  );
  let ( str : any-type = read-input())
    (if lisp::numberp(str) then                % if a number was read,
       str <- lisp::princ-to-string(str)       % convert it to a string
    );
    (while ~lisp::stringp(str) do
      format(t, "~%Invalid input, try again: ");
      str <- read-input()
    );
    str

%-------------------------------------------------------------------------
function Read-Integer(Prompt : string) : integer =
  (if ~empty(Prompt) then
    format(t, "~A:", Prompt)
  );
  let ( int : integer = read-input())
    (while ~lisp::integerp(int) do
      format(t, "~%Invalid input, try again: ");
      int <- read-input()
    );
    int

%-------------------------------------------------------------------------
function Read-Real(Prompt : string) : real =
  (if ~empty(Prompt) then
    format(t, "~A: ", Prompt)
  );
  let ( real-num : real = read-input())
    (while ~lisp::floatp(real-num) do
      format(t, "~%Invalid input, try again: ");
      real-num <- read-input()
    );
    real-num
%-------------------------------------------------------------------------
function Read-Symbol(Prompt : string) : symbol =
  (if ~empty(Prompt) then
    format(t, "~A: ", Prompt)
  );
  let ( sym : string = read-input())
    (while ~lisp::stringp(sym) do
      format(t, "~%Invalid input, try again: ");
      sym <- read-input()
    );
    string-to-symbol(lisp::string-upcase(sym), "RU")
  % NOTE: I convert the string to upper case so that it can be compared to
  % other symbols string-to-symbol returns a symbol that is case sensitive
  % (it is quoted by |'s)
```

```
%-------------------------------------------------------------------------
function Read-Boolean(Prompt : string) : boolean =
  (if ~empty(Prompt) then
    format(t, "~A: ", Prompt)
  );
  format(t,  "(T/t for true, F/f for false): ");
  let(t-or-f : string = read-input())
    (while ~(t-or-f in {"F", "f", "T",  "t"}) do
      format(t, "~%Invalid input, try again: ");
      t-or-f <- read-input()
    );
   t-or-f in {"T", "t"}

%-------------------------------------------------------------------------
function Read-Any-Type(Prompt : string) : any-type =
  (if ~empty(Prompt) then
    format(t, "~A: ", Prompt)
  );
  read-input()

%-------------------------------------------------------------------------

function Read-Yes-Or-No(Prompt : string) : boolean =
  lisp::y-or-n-p(prompt)



%%---------------------- Read Functions With Defaults----------------------
%%% Read functions that allow for a default value

%-------------------------------------------------------------------------
function Read-String-Default(Prompt : string, Default : string) : string =
  (if ~empty(Prompt) then
    format(t, "~A", Prompt)
  );
  format(t, " (~A): ", default);
  let ( str : any-type = read-input())
    (if lisp::numberp(str) then               % if a number was read,
       str <- lisp::princ-to-string(str)      % convert it to a string
    );
    (while ~stringp(str) and ~(lisp::equal(str, Null-Value))  do
      format(t, "~%Invalid input, try again: ");
      str <- read-input()
    );
    if lisp::equal(str, Null-value) then
      Default
    else
      str
```

```
 % For reading in integers and real numbers, read in as a string (so it
 % can be compared to the null-value and check if the string is really an
 % integer or real number (using the read-from-string function).
 % Read-from-string returns two values, the first is what's in the
 % string, the second is the index of the first character NOT read.

%-------------------------------------------------------------------------
function Read-Integer-Default(Prompt : string,
                              Default : integer) : integer =
  (if ~empty(Prompt) then
    format(t, "~A:", Prompt)
  );
  format(t, " (~d): ", default);
  let ( int : string = lisp::read-line())
    (while ~lisp::integerp(lisp::read-from-string(int))  and
                    int ~= Null-Value do
      format(t, "~%Invalid input, try again: ");
      int <- lisp::read-line()
    );
    if int = Null-value then
      Default
    else
      lisp::read-from-string(int)

%-------------------------------------------------------------------------
function Read-Real-Default(Prompt : string, Default : real) : real =
  (if ~empty(Prompt) then
    format(t, "~A: ", Prompt)
  );
  format(t, " (~d): ", default);
  let ( real-num : string = lisp::read-line())
    (while ~lisp::floatp(lisp::Read-from-string(real-num))
                and real-num ~= Null-Value do
      format(t, "~%Invalid input, try again: ");
      real-num <- lisp::read-line()
    );
    if real-num = Null-value then
      Default
    else
      lisp::read-from-string(real-num)

%-------------------------------------------------------------------------
function Read-Symbol-Default(Prompt : string, Default : symbol) : symbol =
  (if ~empty(Prompt) then
    format(t, "~A: ", Prompt)
  );
  format(t, " (~A): ", default);
  let ( sym : string = read-input())
    (while ~lisp::stringp(sym) and sym ~= Null-Value do
      format(t, "~%Invalid input, try again: ");
```

```
    sym <- read-input()
  );
  if sym = Null-value then
    Default
  else
   string-to-symbol(lisp::string-upcase(sym), "RU")
% NOTE: I convert the string to upper case for same reason described in
% Read-Symbol function

%-------------------------------------------------------------------------
function Read-Boolean-Default(Prompt : string,
                              Default : boolean) : boolean =
  (if ~empty(Prompt) then
    format(t, "~A ", Prompt)
  );
  format(t,  "(T/t for true, F/f for false:) ");
  format(t, " (~A): ", default);
  let(t-or-f : string = read-input())
    (while ~(t-or-f in {"F", "f", "T",  "t"}) and t-or-f ~= Null-Value do
      format(t, "~%Invalid input, try again: ");
      t-or-f <- read-input()
    );
    if t-or-f = Null-value then
      Default
    else
   t-or-f in {"T", "t"}

%-------------------------------------------------------------------------
function Read-Any-Type-Default(Prompt : string,
                               Default : any-type) : any-type =
  (if ~empty(Prompt) then
    format(t, "~A: ", Prompt)
  );
  format(t, " (~A): ", default);
  let (ans : any-type =   read-input())

    if lisp::equal(ans, Null-value) then
        % use the lisp::equal incase of strings
      Default
    else
      ans
```

Bibliography

1. Anderson, Captain Cynthia G. *Creating and Manipulating Formalized Software Architectures to Support a Domain-Oriented Application Composition System.* MS thesis, AFIT/GCS/ENG/92D-01, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1992.
2. Bailor, Major Paul D. "Domain-Specific Modeling," *special study class* (April 1992).
3. Barstow, David R. "Domain-Specific Automatic Programming," *IEEE Transactions on Software Engineering*, *11*:1321– 1326 (November 1985).
4. Batory, Don and Sean O'Malley. *The Design and Implementation of Hierarchical Software Systems with Reusable Components.* Technical Report TR-91-22, Austin, Texas: University of Texas, January 1992.
5. D'Ippolito, Richard and Kenneth Lee. *Modeling Software Systems by Domains.* Technical Report, Software Engineering Institute, April 1992.
6. D'Ippolito, Richard S. "Using Models in Software Engineering." *Proceedings: TRI-Ada '89*. 256–265. 1989.
7. D'Ippolito, Richard S. and Charles P. Plinta. "Software Development Using Models," *ACM Sigsoft Software Engineering Notes* (October 1989).
8. Fischer, Charles N. and Richard J. LeBlanc, Jr. *Crafting a Compiler with C.* Redwood City, CA: Benjamin/Cummings Publishing Company, Inc, 1991.
9. Greenspan, Sol J. *Requirements Modeling: A Knowledge Representation Approach to Software Requirements Definition.* PhD dissertation, University of Toronto, Toronto, Ontario, Canada, 1984.
10. Iscoe, Neil. "Domain Modeling – Evolving Research." *Proceedings of the Sixth Annual Knowldege-Based Software Engineering Conference*. 300 – 304. 1991.
11. Iscoe, Neil Allen. *Domain-Specific Programming: An Object-Oriented and Knowledge-based Approach to Specification and Generation.* PhD dissertation, The University of Texas at Austin, Austin Texas, 1990.
12. Kang, Kyo C. and others. *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, November 1990 (AD-A235 785).
13. Kelly, Van E. and Uwe Nonnenmann. "Reducing the Complexity of Formal Specification Acquisition." *Automating Software Design* edited by Michael R. Lowry and Robert D. McCartney, Menlo Park, CA: AAAI Press, 1991.
14. Korth, Henry F. and Abraham Silberschatz. *Database System Concepts, 2nd edition.* New York, NY: McGraw-Hill, Inc., 1991.
15. Lee, Kenneth J. and others. *Model-Based Software Development (Draft).* Technical Report CMU/SEI-92-SR-00, Software Engineering Institute, December 1991.

16. Lockheed Software Technology Center, Palo Alto, CA. *Software User's Manual for the Automatic Programming Technologies For Avionics Software (APTAS) System*, June 1991.

17. Lowry, Michael R. "Software Engineering in the Twenty-first Century." *Automating Software Design*, edited by Michael R. Lowry and Robert D. McCartney. 627–654. Menlo Park, CA: AAAI Press, 1991.

18. Neighbors, James M. "The Draco Approach to Constructing Software from Reusable Components," *IEEE Transactions on Software Engineering*, *10*:564–574 (September 1984).

19. Neighbors, James M. "Draco: A Method for Engineering Reusable Software Systems." *Tutorial: Domain Analysis and Software Systems Modeling* edited by R. Prieto-Diaz and G. Arango, Los Alamitos, CA: IEEE Computer Society Press, 1991.

20. Peterson, A. Spencer. "Coming to Terms with Software Reuse: A Model-based Approach," *ACM SIGSOFT Software Engineering Notes*, *16*:45–51 (April 1991).

21. Prieto-Díaz, Rubén. "Domain Analysis: An Introduction," *ACM SIGSOFT Software Engineering Notes*, *15*:47–54 (April 1990).

22. Prieto-Díaz, Rubén. "Domain Analysis for Reusability." *Proceedings of the 11th Annual International Computer Software and Application Conference*. 23–29. IEEE Computer Society Press, 1990.

23. Reasoning Systems, Inc. *DIALECT User's Guide*. Palo Alto, CA, July 1990.

24. Reasoning Systems, Inc. *REFINE User's Guide*. Palo Alto, CA, May 1990.

25. Reubenstein, Howard B. and Richard C. Waters. "The Requirements Apprentice: an Initial Scenario." *Proceedings of the Fifth International Workshop on Software Specification and Design*. 211–218. May 1989.

26. Smith. "KIDS: A Semiautomatic Program Development System," *IEEE Transactions on Software Engineering*, *16*:1024–1043 (September 1990).

27. Smith, Douglas R. "KIDS – A Knowledge-Based Software Development System." *Automating Software Design*, edited by Michael R. Lowry and Robert D. McCartney. Chapter 19. Menlo Park, CA: AAAI Press/MIT Press, 1991.

28. Spicer, Kelly L. *Mapping an Object-Oriented Requirements Analysis to a Design Architecture that Supports Reuse*. MS thesis, AFIT/GCS/ENG/90D, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1990.

29. Weide, Lieutenant Timothy. *Visualization and Manipulation of a Formal Object Base (Draft)*. MS thesis, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, March 1993.

30. Zeroual, K. "KBRAS: A Knowledge-based Requirements Acquisition System." *Proceedings of the Sixth Annual Knowldege-Based Software Engineering Conference*. 40–52. 1991.

Vita

Mary Anne Randour was born May 8, 1964 in Cape May NJ. She graduated from Lower Cape May Region High School in 1982 and then went to Lehigh University on a four-year Air Force Reserve Officer Training Corps scholarship. Upon graduation in 1986, she was commissioned a Second Lieutenant in the Air Force. She was assigned to the 1912th Computer Systems Group at Langley AFB. In May 1991, she entered the Air Force Institute of Technology to pursue a Master of Science degree in Computer Science.

Permanent address: 1112 Missouri Ave
Cape May NJ 08204

REPORT DOCUMENTATION PAGE

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1992	Master's Thesis

4. TITLE AND SUBTITLE

CREATING AND MANIPULATING A DOMAIN-SPECIFIC FORMAL OBJECT BASE TO SUPPORT A DOMAIN-ORIENTED APPLICATION COMPOSITION SYSTEM

6. AUTHOR(S)

Mary Anne Randour, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCS/ENG/92D-13

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

ASC/RWWW
Wright-Patterson AFB, OH 45433-6583

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This research investigated technology which enables sophisticated users to specify, generate, and maintain application software in domain-oriented terms. To realize this new technology, a development environment, called Architect, was designed and implemented. Using canonical formal specifications of domain objects, Architect rapidly composes these specifications into a software application and executes a prototype of that application as a means to demonstrate its correctness before any programming language specific code is generated. This thesis investigated populating and manipulating the formal object base required by Architect. This object base is built using a domain-specific language (DSL) which serves as an interface between the user and a domain model. The domain model describes primitive domain object classes and composition rules and is formalized via a domain modeling language. The packaging of the objects into components is defined by an architecture model which was part of a separate thesis. The Software Refinery environment was used to develop a methodology for defining DSLs for Architect and for manipulating the resulting populated object base. Architect was validated using both artificial and realistic domains and was found to be a solid foundation for an application generation system.

14. SUBJECT TERMS

computers, computer programs, software engineering, specifications, domain-specific languages, domain modeling, application composition systems, software architecture models

15. NUMBER OF PAGES

212

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL